VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Kvadrantové stromy a jejich varianty

Quad-trees and its Variants

2012

Bc. Tomáš Plinta

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student:	**Bc. Tomáš Plinta**
Studijní program:	N2647 Informační a komunikační technologie
Studijní obor:	2612T025 Informatika a výpočetní technika
Téma:	Kvadrantové stromy a jejich varianty Quad-trees and its Variants

Zásady pro vypracování:

Cílem je naimplementovat a otestovat datovou strukturu kvadrantových stromů a jejich variant. Součástí práce bude vyhodnocení a srovnání výsledků pro implementované datové struktury.

1. Nastudujte problematiku kvadrantových stromů a jejich variant.
2. Navrhněte a implementujte tyto datové struktury.
3. Proveďte otestování datových struktur.
4. Proveďte vyhodnocení výsledků a srovnání s dalšími datovými strukturami.

Seznam doporučené odborné literatury:

Podle pokynů vedoucího bakalářské práce.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Filip Křižka**

Datum zadání: 18.11.2011
Datum odevzdání: 04.05.2012

VYSOKÁ ŠKOLA BÁŇSKÁ-TECHNICKÁ UNIVERZITA Fakulta elektrotechniky a informatiky OSTRAVA

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlášení studenta

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě dne 2.5.2012 .....

.....

Tomáš Plinta

Poděkování

Rád bych poděkoval Ing. Filipu Křižkovi a doc. Ing. Michalu Krátkému, Ph.D. za odbornou pomoc a konzultaci při vytváření této diplomové práce.

Abstrakt

Tato diplomová práce se zabývá výzkumem a praktickou implementací vícedimenzionálních datových struktur se zaměřením na kvadrantové stromy. Nejprve je teoreticky popsáno, na jakém principu pracují, jejich vlastnosti a varianty. V další části je popsán postup konkrétní implementace kvadrantových stromů. Dále jsou uvedeny problémy, se kterými jsem se setkal a také jejich řešení. Poté jsou provedeny výkonové testy a porovnání s dalšími datovými strukturami. Konec práce obsahuje jak zhodnocení testů, tak celkové zhodnocení diplomové práce, získané znalosti a dovednosti.

Klíčová slova

kvadrantové stromy, databáze, indexování, multidimenzionální datové struktury

Abstract

This diploma thesis deals with research and practical implementation of multidimensional data structures, with focus on quadtrees. First, it is theoretically described, on what principle is used, their properties and variants. Next section describes procedure of implementation of quadtree. Hereafter, issues, which were occurred during implementation are described, together with their solution. Afterwards, performance tests are performed and quadrees are compared with other data structures. End of this thesis includes evaluation of tests and overall diploma thesis, used and gained knowledge and skills.

Key words

quadtrees, database, indexing, multidimensional data structures

Seznam použitých zkratek

Zkratka	Anglický význam	Český význam
d	dimension	dimenze prostoru
DBRG	Database Research Group	Databázová výzkumná skupina
NW	North West	Severozápadní
SW	South West	Jihozápadní
NE	North East	Severovýchodní
SE	South East	Jihovýchodní
MBR	Minimum bounding rectangle	Minimální ohraničující obdélník

Obsah

1 Úvod

V souvislosti s rozvojem informačních technologií souvisí otázka, jak efektivně přistupovat k množství uložených dat. V případě vícedimenzionálních dat už není možné jednoduše přistupovat k datům sekvenčně, zvláště v případech, kdy výstupem dotazu má být relativně malá množina prvků. V případě, kdy to povaha dat umožňuje, je vhodné vytvořit si nad daty strukturu, díky které se při vyhledávání požadovaných dat projde malá množina prvků vzhledem k celkové množině dat.

Takovéto strukturalizování dat se nazývá indexování. Hlavní parametr při tvorbě indexů je, jak dokáže daná indexová struktura zefektivnit vyhledávání dat a celkově zrychlit práci s daty. Je však vhodné dbát i na to, aby výsledná velikost daného indexu nepřesáhla velikost původního datového souboru. Dále je třeba mít na paměti, aby režie daného indexu, čili jak je časově náročné procházet indexovou strukturou, nepřesáhla únosnou mez. Pokud je problém v jednom z předchozích bodů, je vhodné aplikovat jiný typ indexu. Proto je vhodné znát charakter dat a podle něj určit, jaký typ indexu je pro daná data nejvhodnější.

V případě dat, která uchovávají prostorovou složku, už běžné indexové struktury nestačí. Je třeba využít jednu z mnoha metod pro indexování vícerozměrných dat. Mezi ně patří široké spektrum struktur, které mají odlišné chování.

Tato diplomová práce se zabývá především kvadrantovými stromy, které využívají principů známých z binárních vyhledávacích stromů.

Ve druhé kapitole jsou krátce popsána jednorozměrná data, přičemž hlavní část kapitoly vysvětluje problematiku vícerozměrných datových struktur, jejich efektivitu a jakým způsobem je možné z nich získávat data.

Třetí kapitola pojednává už pouze o kvadrantových stromech, je zde přiblížen princip, na jakém pracují, kdy je a kdy není vhodné je použít. Taktéž je podrobně popsána práce s uzly při vkládání, odstraňování a vyhledávání.

Ve čtvrté kapitole přibližuji problematiku R-stromů, protože s touto strukturou se budou kvadrantové stromy porovnávat po výkonové stránce.

V páté kapitole se věnuji implementaci kvadrantových stromů do frameworku DBRG. Do detailů jsou zde popsány jednotlivé třídy a jejich metody.

V šesté kapitole jsou provedeny výkonnostní testy a jejich analýza. Dále je realizováno porovnání s další datovou strukturou – R-stromem.

V poslední, sedmé kapitole jsou shrnuty dosažené výsledky, použité a nabyté zkušenosti.

2 Vícerozměrné datové struktury

V následující kapitole je rozebrána problematika jednorozměrných i vícerozměrných dat. Obecně je zde popsáno, na jakém principu fungují, jaká je jejich efektivita a jakým způsobem je možné z nich získávat data.

2.1 Jednorozměrná data

Mezi jednorozměrná data patří běžné typy dat, které známe a se kterými se denně setkáváme, ať už při programování, či jako uživatelé. Patří zde struktury a datové typy, které používají k popisu nějaké vlastnosti pouze jeden atribut, čili například identifikační číslo, věk, přihlašovací jméno apod. Mezi jednorozměrná data patří i adresa, která ač se může skládat z více údajů jako město, ulice, stát, stále je to jeden atribut. Naopak mezi jednorozměrné struktury již nepatří například slovníkové struktury z vyšších programovacích jazyků, kde se jeden záznam skládá z údajů klíč - hodnota.

2.2 Vícerozměrná data

Dalším typem dat jsou vícerozměrná data. Reprezentace vícerozměrných datových struktur je v dnešní době v souvislosti s databázovými systémy velmi diskutované téma. Mají široké využití v počítačové grafice, vizualizaci, zpracování obrazu nebo geoinformatice. Datové prvky v těchto strukturách mohou reprezentovat jak umístění nebo objekty v prostoru, tak i obecné záznamy. Jako typický příklad vícerozměrných datových struktur, reprezentující obecné záznamy můžeme uvést atributy jméno, věk, pohlaví, adresa nebo datum narození u typu entity student. Tyto záznamy s uvedenými atributy, ač jsou jednorozměrné, můžou být v systému řízení báze dat interpretovány jako vícerozměrné. V tomto konkrétním případě tedy jako pětirozměrné (pro každý atribut jeden rozměr). Každý atribut těchto dat je reprezentován jiným datovým typem (například atributy jméno a adresa jsou reprezentovány řetězcem, pohlaví booleovskou hodnotou nebo znakem, věk číselnou hodnotou a datum narození datovým typem datum) [1].

Dalším typem vícerozměrných dat jsou data o umístění prvků v prostoru. Tyto data obsahují navíc jednu důležitou vlastnost a to je vzdálenost prvků v prostoru [1]. Samozřejmě platí, že všechny atributy reprezentující souřadnice prvku v prostoru musí mít stejný typ jednotek. Takovýmto datům obecně říkáme metrická data. Jako příklad metrických dat zavedeme následující množinu dat reprezentující geografické umístění prvků v prostoru (Tabulka 2.1). Pro názornost a snadnou představitelnost uvedeme pouze data s dimenzí d=2.

Název	*X*	*Y*
Praha	*30*	*40*
Brno	*55*	*24*
Ostrava	*67*	*66*
Plzeň	*74*	*77*
Olomouc	*13*	*54*
Liberec	*25*	*42*
Zlín	*73*	*12*
Kladno	*94*	*10*

Příklad reprezentace dvourozměrných dat. 2.1

Z této množiny dat je patrné, že reprezentují lokaci měst v prostoru (pozn.: souřadnice neodpovídají reálným hodnotám). Každý bod (město) v prostoru je zde reprezentován dvěma atributy *x* a *y*. Součástí každého záznamu je také textový popis bodu, který ale neslouží pro lokaci jednotlivých bodů, takže dimenze těchto typů záznamů zůstává rovna dvěma.

2.3 Typy dotazů

U jednorozměrných dat jsou dotazy poměrně jednoduché, a to pouze zda je záznam s danou hodnotou atributu v datovém souboru či nikoliv, respektive zda existuje záznam, u kterého je hodnota určitého atributu v daném rozmezí hodnot. U vícerozměrných dat nás opět může zajímat, zda záznam s daným souborem hodnot leží či neleží v dané datové struktuře. Tento dotaz se nazývá bodový dotaz (point query).

Dále nás může zajímat, jaké body leží v daném rozsahu hodnot, čili u dvourozměrných dat jaké body leží uvnitř dané plochy. Tento typ dotazu se nazývá rozsahový dotaz (range query). Speciálním typem rozsahového dotazu pak může být dotaz na vyhledání prvků v dané ploše, přičemž střed této plochy bude v nějakém bodu P, ležícím v dané struktuře.

Můžeme mít také určitý bod v prostoru a zajímá nás vyhledání *N* nejbližších bodů. Takový dotaz patří do kategorie vyhledání nejbližších sousedů (Nearest neighbor search). Tento dotaz je výpočetně nejnáročnější a je obdobou post office problému popsaném Donaldem Knuthem [9].

2.4 Vícerozměrné datové struktury

Datových struktur pracujících s vícerozměrnými daty je relativně velké množství, liší se jednak typem zpracování vícerozměrných dat, tak také samozřejmě výpočetní složitostí. Některé struktury pracují na principu určitého zpracování vstupního datového souboru jako například invertovaný soubor, nicméně velká většina však vícedimenzionální data ukládá nějakým způsobem do datové struktury typu strom.

Mezi struktury pracující s vícerozměrnými daty patří například bitmapová reprezentace. Tato reprezentace je však vhodná pouze tehdy, pokud je počet atributů p velmi malý ($p \leq 2$) a atributy jsou diskrétního typu s malou doménou (například binární hodnoty).

Jedna z nejjednodušších reprezentací vícerozměrných dat je sekvenční list. Jak název napovídá, data zde nejsou uložena v žádné sofistikované datové struktuře, ale leží sekvenčně za sebou. Není zde aplikováno seřazení pro žádný atribut. Z tohoto vyplývá, že při N záznamech a d rozměrech je složitost vyhledání určitého záznamu $O(N * d)$ [1] v nejhorším případě, jelikož se musí projít všechny záznamy v datovém souboru. Tato struktura tedy není příliš vhodná pro reprezentaci vícedimenzionálních dat.

Další reprezentaci představuje invertovaný soubor [1]. Invertovanost spočívá v tom, že se obrátí úloha záznamů a atributů. Nesledujeme tedy hodnoty atributů pro jednotlivé záznamy, ale naopak všechny záznamy, které obsahují určitou hodnotu daného atributu. Dle tohoto atributu jsou následně záznamy seřazeny, pokud se jedná o numerické či alfanumerické hodnoty. Z předchozího tedy vyplývá, že pro každý atribut existuje právě jeden invertovaný list. Záznamy v jednotlivých invertovaných listech jsou ve skutečnosti ukazatele na daný záznam. Z invertovaného listu vychází multilist, kde je ke každému záznamu vytvořeno tolik linkovacích seznamů, kolik existuje atributů. Postup pro vytvoření invertovaného souboru pro data z Tabulky 2.1 je následující:

1. Nejprve se uloží seznam měst a jejich souřadnice x.
2. Následně se seřadí dané x-ové souřadnice
3. Dojde k nahrazení x-ových souřadnic danými městy, ležících na těchto x-ových souřadnicích.
4. Tento postup se opakuje pro každý další atribut - souřadnici.

Vlastností tohoto přístupu je zrychlení vyhledávání daného prvku za cenu zvýšení počtu operací při vkládání nových prvků. Při každém vložení nového záznamu je totiž třeba vložit prvek i do jednotlivých invertovaných listů a provést jejich seřazení. Je tedy zřejmé, že s rostoucí dimenzí dat efektivita toho přístupu klesá.

X	*Y*
Olomouc	*Kladno*
Liberec	*Zlín*
Praha	*Brno*
Brno	*Praha*
Ostrava	*Liberec*
Zlín	*Olomouc*
Plzeň	*Ostrava*
Kladno	*Plzeň*

Výsledný invertovaný soubor pro data z Tabulky 2.1 2.2

Použití datové struktury invertovaného souboru je velmi vhodné, pokud budeme primárně vyhledávat podle jednoho klíče (atributu). V takovém případě využijeme invertovaný soubor pro daný atribut. Při vyhledávání v tomto souboru lze s úspěchem využít efektivnější metody vyhledávání než sekvenční průchod a to například metodou půlení intervalu. Tato metoda totiž předpokládá, že daný seznam hodnot (zde tedy invertovaný soubor) bude seřazen. Nejprve najde medián daného souboru a porovná jej s hledanou hodnotou. Pokud medián není hledaná hodnota, pokračuje stejným způsobem v levé nebo pravé části souboru na základě toho, zda je hledaná hodnota větší či menší než vyhledaný medián. Složitost tohoto vyhledání je oproti sekvenčnímu průchodu se složitostí $O(N)$ logaritmická $O(\log N)$ [2].

Při použití vyhledání zahrnujícího více atributů, případně všechny, však výpočetní složitost narůstá a například při rozsahových dotazech dosahuje složitosti $O(N^{\left(1-\frac{1}{d}\right)})$, kde d označuje počet atributů datového souboru [1]. Z toho vyplývá, že použití této struktury při rozsahových dotazech pracujících s více atributy je značně neefektivní. Složitost se dá částečně snížit, pokud provedeme rozsahový dotaz pro každý atribut zvlášť a výsledek bude průnik těchto parciálních rozsahových dotazů. Jiným přístupem je použití kombinovaného indexu. Kombinovaný index je variantou invertovaného souboru s tím rozdílem, že jednotlivé soubory pro atributy jsou seřazeny v abecedním pořadí.

2.4.1 Struktury založené na hodnotách atributů

V této skupině vícedimenzionálních datových struktur jsou zahrnuté ty, které pracují přímo s hodnotami jednotlivých atributů.

Mezi ně patří multiatributový strom [1]. U této reprezentace se předpokládá, že všechny hodnoty jsou lexikograficky uspořádány podle pořadí atributů. To znamená, že v prvním atributu jsou hodnoty seřazeny bez omezení, ve druhém atributu jsou následně seřazeny ty hodnoty, které mají stejnou hodnotu prvního atributu. Tímto postupem se pokračuje pro všechny zbývající atributy. Pro příklad je uvedena tabulka 2.3, která obsahuje neseřazený datový soubor. Tabulka 2.4 pak obsahuje seřazené hodnoty atributů podle výše uvedené metody.

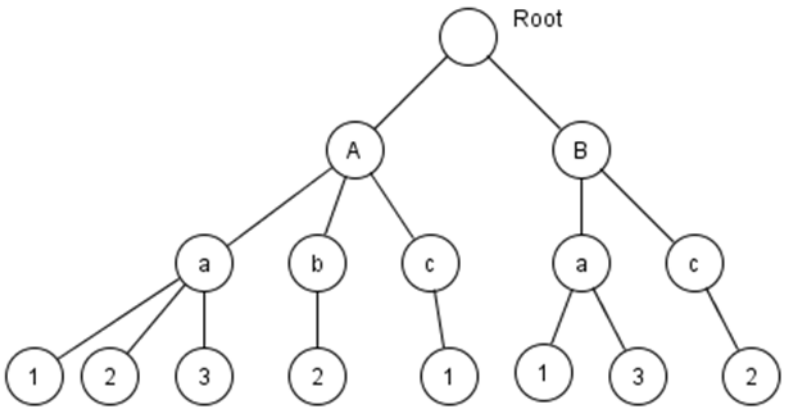
Atribut 1	*Atribut 2*	*Atribut 3*
B	*c*	*2*
A	*a*	*2*
A	*b*	*2*
B	*a*	*1*
A	*a*	*1*
B	*a*	*3*
A	*c*	*1*
A	*a*	*3*

Původní neseřazený datový soubor. 2.3

Atribut 1	*Atribut 2*	*Atribut 3*
A	*a*	*1*
A	*a*	*2*
A	*a*	*3*
A	*b*	*2*
A	*c*	*1*
B	*a*	*1*
B	*a*	*3*
B	*c*	*2*

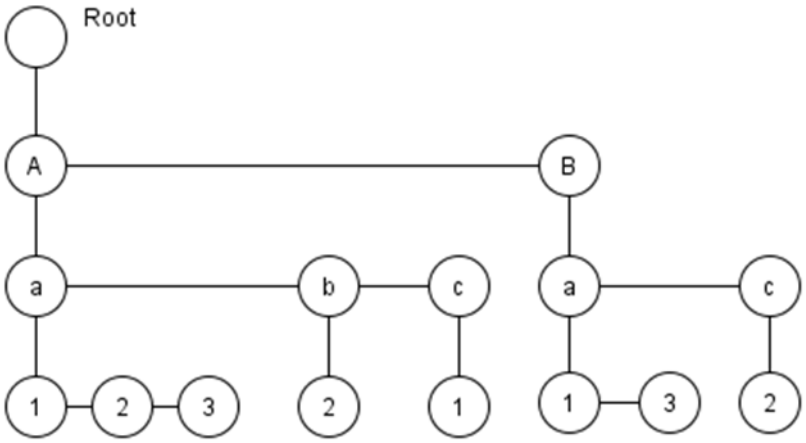
Seřazený datový soubor. 2.4

Data na obrázku 2.1 pak ilustrují způsob uložení záznamů v multiatributovém stromu. Z kořene vedou hrany ke všem unikátním hodnotám prvního atributu. V našem případě A a B. Tyto dvě hodnoty tedy reprezentují dva uzly stromu, ležící ve stejné hloubce. Z těchto dvou uzlů pak dále vedou další hrany ke všem unikátním hodnotám druhého atributu. Toto se opakuje až po poslední atribut, z čehož vyplývá, že počet atributů bude roven hloubce stromu. Při provedení bodového dotazu tedy bude počet nutných kroků roven hloubce stromu. U této struktury tedy záleží i na pořadí atributů. Pokud tedy bude mít první atribut velkou doménu, efektivita se snižuje.



Multiatributový strom pro zdrojová data z tabulky 2.4 2.1

Velmi podobnou strukturou je DCT (doubly chained) strom [1]. Tento strom vychází z multiatributového stromu a liší se od něj v několika aspektech. Z každého uzlu vedou nanejvýš dvě hrany. Jedna z hran vede k potomkovi, pokud daný uzel nějaký má a druhá vede k sousednímu uzlu na dané úrovni. Pro DCT platí stejné pravidlo jako u multiatributového stromu a to, že hloubka stromu bude rovna počtu atributů.



DCT strom pro zdrojová data z tabulky 2.4 2.2

Existuje mnoho dalších vícedimenzionálních datových struktur patřících do této kategorie, jako jsou vícedimenzionální B-stromy, quintary stromy, kB stromy, nicméně účelem této diplomové práce není popsání veškerých existujících datových struktur, tudíž nyní přejdeme k další kategorii vícerozměrných datových struktur.

2.4.2 Struktury založené na vícerozměrných buňkách

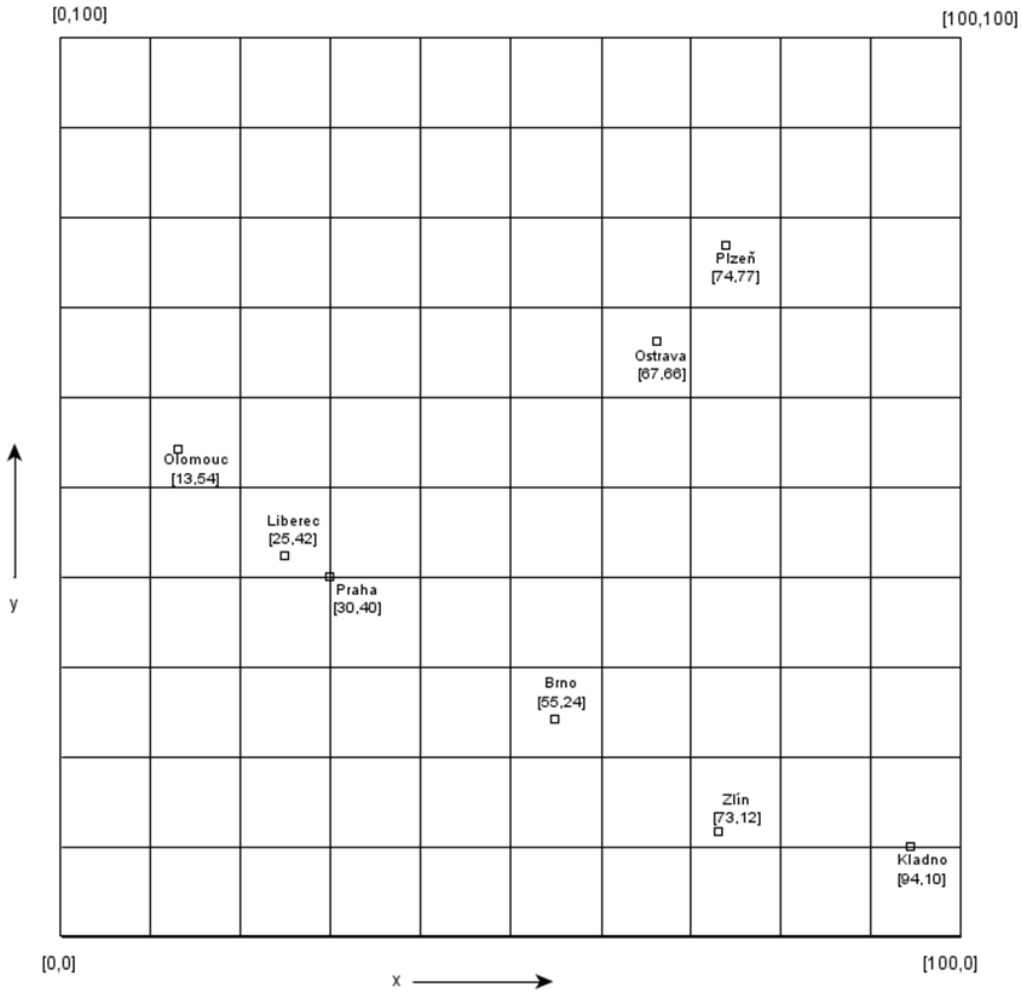
Zde patří vícedimenzionální datové struktury, které rozdělují prostor do vícerozměrných buněk. Pokud hodnoty určitého záznamu leží v rozsahu vymezeném danou buňkou, jsou k ní asociovány. Každá buňka poté obsahuje ukazatel na jinou datovou strukturu, ve které jsou uloženy všechny body, které leží v dané buňce.

Na tomto principu je založena metoda fixed-grid [1], která pracuje s buňkami pevné délky. Plocha, na které se nacházejí jednotlivé body, je rozdělena do čtvercových buněk stejné velikosti za jakýchkoliv okolností a nazývá se uniformní mřížka. Každý bod ležící v této mřížce je pak vyhledán v konstantním čase. Každá jednotlivá buňka pak obsahuje ukazatel na pole, kde jsou uloženy dané body. U metody fixed grid tedy nezáleží na pořadí atributů, ale pouze na rovnoměrném rozmístění dat (bodů). Pokud tedy máme většinu bodů převážně v jedné buňce, výpočetní složitost při bodovém dotazu se blíží sekvenčnímu průchodu polem $O(N)$. Pokud jsou však data přibližně rovnoměrně rozmístěna, stává se tato metoda relativně efektivní.

Při bodovém dotazu se tedy nejprve najde buňka, ve které by měl daný bod ležet a následně se projde přidružené pole bodů. Pokud zde není, daný bod ve struktuře neexistuje. Při rozsahovém dotazu se musí prohledat veškeré buňky, respektive pole asociované s buňkami, které překrývá daná plocha rozsahového dotazu. Pokud tedy budeme chtít vyhledat body, které leží ve čtverci se středem v bodě [70,70] s délkou strany $r = 20$, musíme prohledat buňky se středy v bodech [65,65], [75,65],

[65,75] a [75,75]. V každém poli asociovaném s buňkou pak musíme u všech bodů zkontrolovat, zda neleží v požadovaném rozsahu. Z toho vyplývá, že složitost tohoto vyhledání je $O(F * 2^d)$ [1], kde F je počet nalezených bodů a d je dimenze bodů. Při větší dimenzi tedy výpočetní složitost exponenciálně narůstá, takže je tato metoda vhodná pouze u nižších dimenzí. Nicméně i u dimenze $d = 2$ musíme prohledat spoustu bodů navíc. U uniformní mřížky s velkým rozsahem buňky a velkým počtem bodů je tato metoda velice neefektivní.

Pokud bod leží na rozhraní dvou buněk, uplatňujeme pravidlo, že každá buňka je otevřená shora a zprava a naopak uzavřená zdola a zleva. Pokud tedy například máme buňku, jejíž levý spodní roh leží v bodě [20,20] a vrchní pravý roh v bodě [40,40], poté bod P1 se souřadnicemi [20,30] bude ležet v této buňce a naopak bod P2 se souřadnicemi [40,30] bude ležet v buňce napravo od této buňky.



Příklad uniformní mřížky se zdrojovými daty z tabulky 2.1 2.3

Metoda fixed grid je vhodná, pokud jsou data rovnoměrně rozptýlena po celém prostoru. Nicméně tento ideální stav není běžný, některé buňky mohou obsahovat mnoho bodů a jiné zase být

prázdné. Pokud je tedy obsah buněk značně nevyvážený, efektivita této metody klesá. Nicméně existuje několik způsobů jak se s tímto stavem vyrovnat.

První způsob stojí na principu spojit sousedící prázdné buňky do větší prázdné buňky a zároveň průběžně rozdělovat příliš plné buňky do několika menších. Toto řešení vede k nutnosti používat místo polí stromové struktury k získání požadovaných bodů. Dostáváme se tedy k hlavnímu obsahu této diplomové práce, protože v tomto případě jsou využívány právě kvadrantové stromy, ať už to jsou bodové kvadrantové stromy, či prefixové stromy jako MX a PR kvadrantové stromy.

3 Kvadrantové stromy

Kvadrantové stromy jsou jednou z nejefektivnějších datových struktur pro ukládání vícedimenzionálních dat [1]. Byly popsány Raphaelem Finkelem a Jonem Luisem Bentleyem v roce 1974 [3]. Někdy se taktéž nazývají Q-stromy. Spojují se v nich principy k-nárního stromu s metodou fixed grid [1]. Ve své podstatě rekurzivně rozdělují dvourozměrný prostor do čtyř kvadrantů. Konkrétní podoba tohoto rozdělování (dekompozice) je dána vstupními daty a pořadím vstupních dat. Využívá se zde také principu, kdy se sousedící prázdné buňky spojují do větších prázdných buněk [3].

Kvadrantové stromy rozdělujeme do několika skupin, zejména podle dat, které reprezentují:

- Bodové kvadrantové stromy (Point Quadtree)
- Prefixové kvadrantové stromy (Trie based quadtrees)
    - MX kvadrantové stromy
    - PR kvadrantové stromy

3.1 Bodové kvadrantové stromy

Bodové kvadrantové stromy vycházejí z principu binárních stromů a používají se k reprezentaci vícerozměrných bodových dat. Tudíž ve své podstatě můžeme říci, že to jsou vícerozměrné binární vyhledávací stromy [1]. S navýšením rozměru také souvisí následující úprava binárních vyhledávacích stromů:

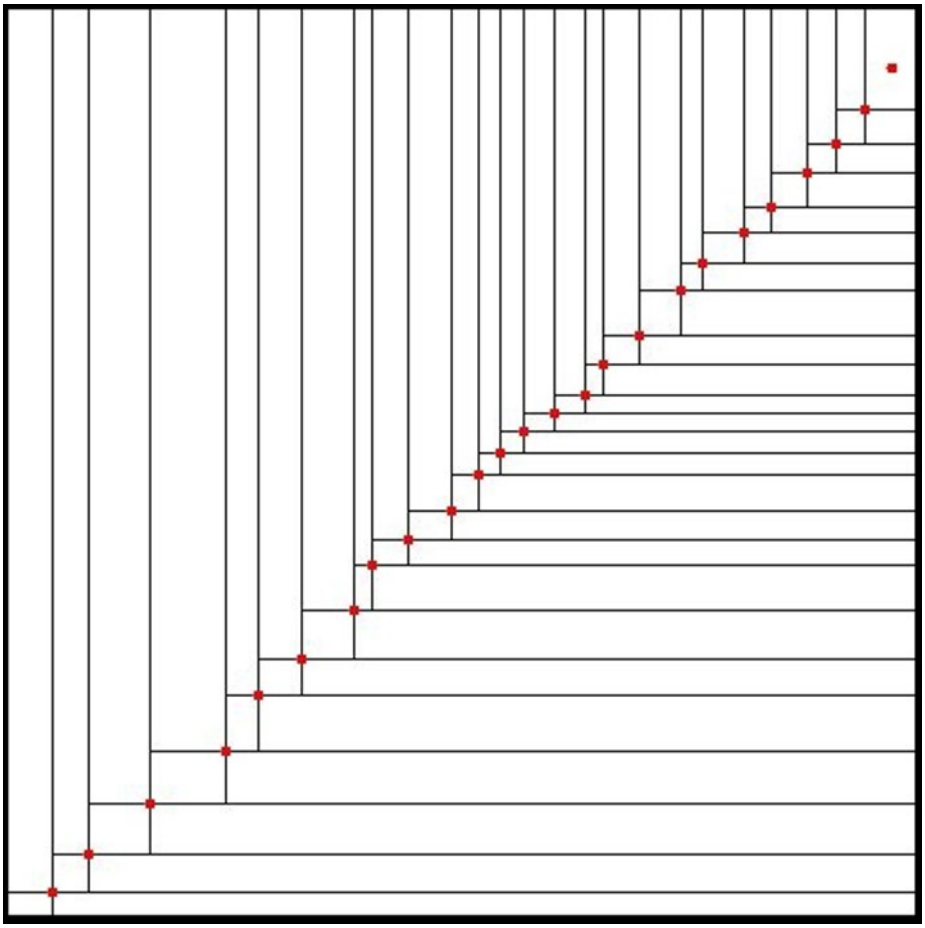
- *Každý uzel, pokud není listem, má právě 2^d potomků, kde d značí dimenzi* [1].

Tito potomci tedy jistým způsobem reprezentují u dvourozměrného kvadrantového stromu čtyři podprostory (kvadranty), do kterých se rozděluje základní prostor, ve kterém leží body.

Práce kvadrantového stromu vypadá následovně. První bod, který je do této struktury vložen, je umístěn na vrchol stromu a slouží jako kořen. Další bod, vkládaný do stromu je vložen do odpovídajícího kvadrantu. Z toho je zřejmé, že záleží na pořadí, ve kterém body do stromu vkládáme. Při jiném pořadí vkládání bodů do stromu se strom může zformovat jinak.

Na začátku této kapitoly bylo zmíněno, že kvadrantové stromy jsou velmi efektivní metodou ukládání vícerozměrných dat. Nicméně, velmi záleží, jakým způsobem jsou body rozloženy v prostoru a v jakém pořadí jsou vkládány. Pokud si představíme případ, kdy máme první bod umístěn na souřadnici [1,1] a následně vkládané body můžeme proložit funkcí $y = x$, budou tedy ležet na přímce. V tomto případě se uloží bod *P1* jako kořen stromu, bod *P2* dva se uloží do druhého kvadrantu bodu 1, bod 3 opět do druhého kvadrantu bodu 2 atd. Zjednodušené znázornění můžeme vidět na obrázku 3.1. V tomto případě se kvadrantový strom stane nevyváženým s velkou hloubkou. Hloubka stromu h dosahuje počtu vložených bodů N. V tomto případě tedy je složitost vyhledávání prvku $O(N)$

v nejhorším případě, což dělá z kvadrantových stromů velice neefektivní strukturu. Nicméně i toto se dá vyřešit. V takovýchto případech se využívají vyvážené kvadrantové stromy [1].



Znázornění datového souboru, pro který je nevhodné použít kvadrantový strom 3.1

3.1.1 Vkládání prvků do bodového kvadrantového stromu

Vkládání prvků do bodového kvadrantového stromu je velice podobné vkládání prvku do binárního vyhledávacího stromu [1].

Pokud je strom prázdný, první prvek se vloží na pozici kořene stromu. Pokud strom není prázdný, provede se vyhledání uzlu se záznamem, který má hodnoty stejné, jako právě vkládaný bod *P*. Pokud je uzel nalezen, provede se přepsání tohoto nalezeného uzlu asociovaného s bodem novým vkládaným bodem *P*. Pokud uzel není nalezen, nacházíme se na neexistujícím uzlu (NIL uzel) a vkládáme na toto umístěn nový uzel s bodem *P*.

Vyhledání samotného uzlu probíhá následujícím způsobem:

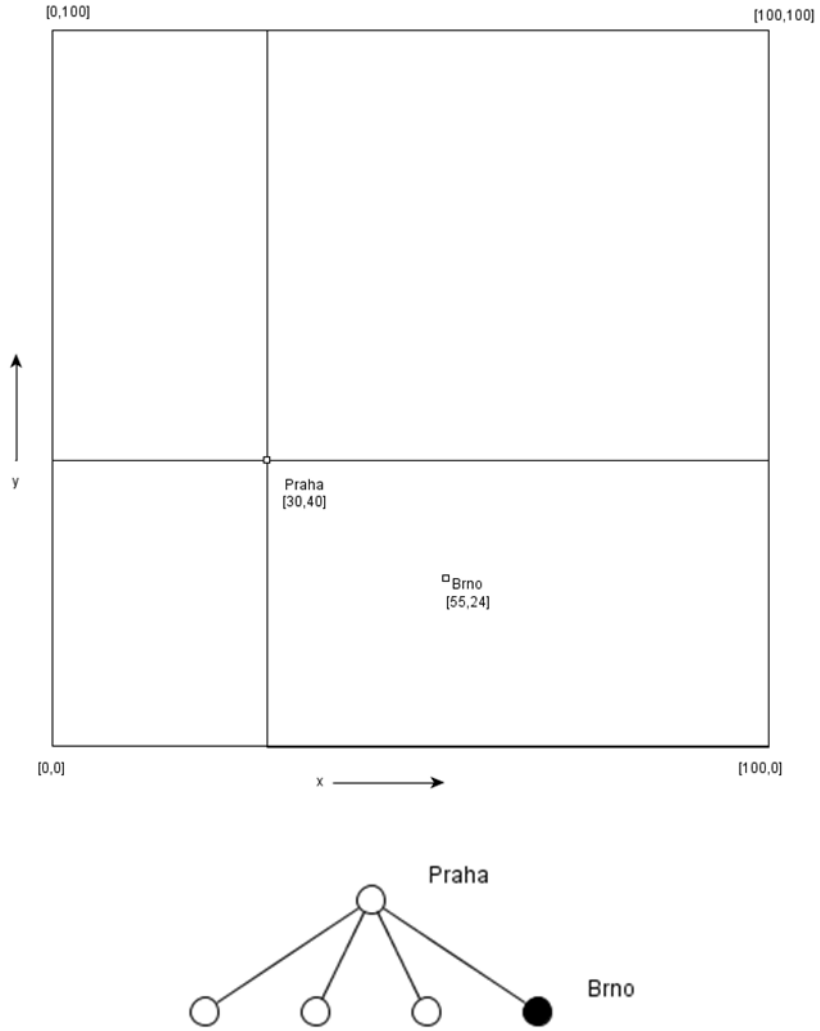
1. Porovnání kořene stromu s vkládaným bodem *P*.
2. Pokud klíče nejsou stejné, pokračuje se dalším potomkem (kvadrantem). Potomek, který se bude porovnávat, se určí podle hodnot klíče. To znamená, že se porovnají souřadnice

X a *Y* bodu *P* a bodu v kořeni stromu. Podle výsledku tohoto porovnání se pak určí, ve kterém kvadrantu se bude pokračovat. Pokud bude například $Px < Kx$ a zároveň $Py > Ky$, kde *Px*, *Py* jsou hodnoty souřadnic *x* či *y* bodu *P* a *Kx* respektive *Ky* hodnoty souřadnic bodu asociovaném v kořeni stromu, pak se bude pokračovat v potomku asociovaném jako I. kvadrant (severozápadní - NW kvadrant).

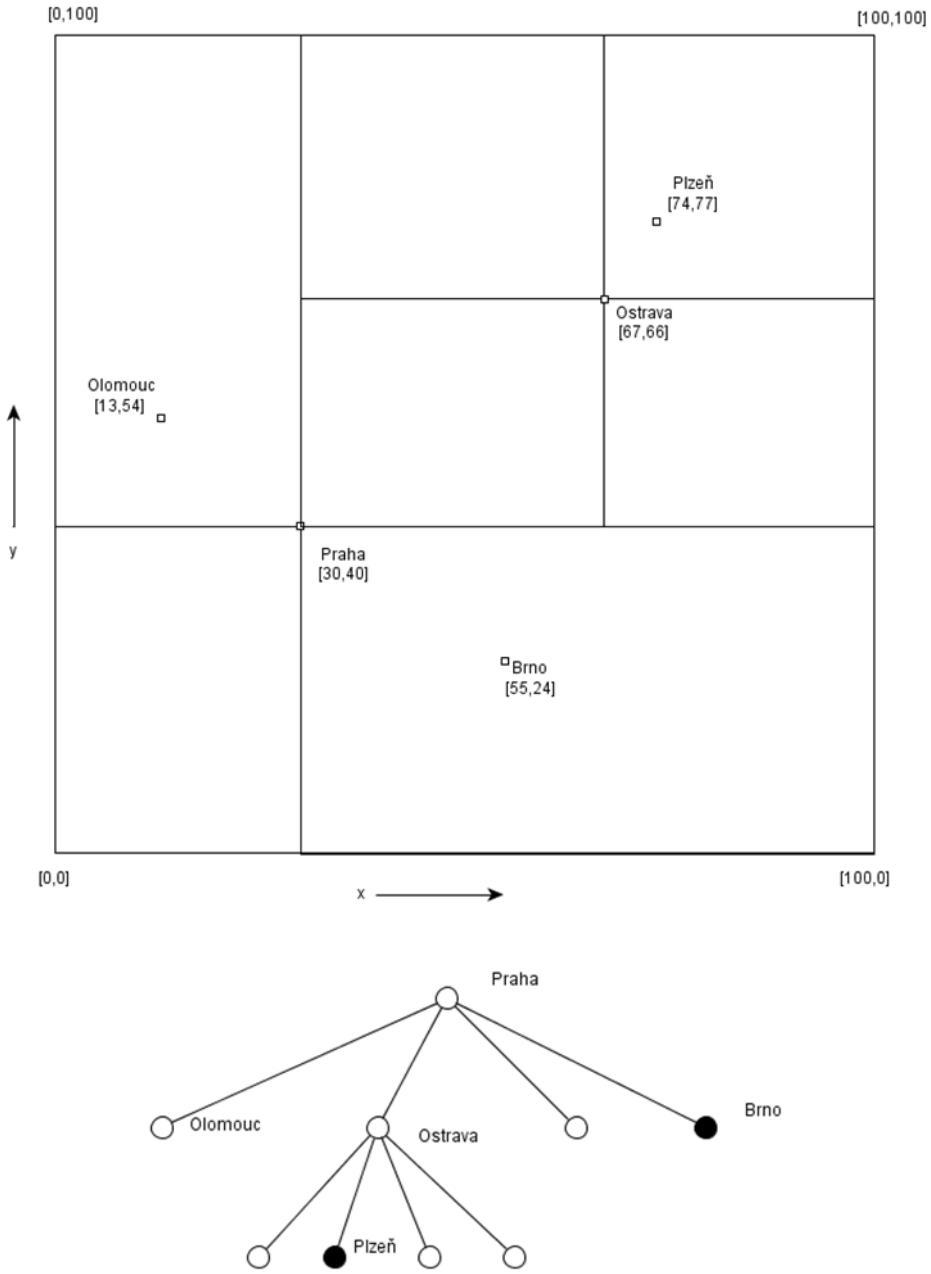
3. Bod P se porovná s bodem asociovaným v tomto uzlu. Pokud hodnoty atributů nejsou stejné, pokračuje se opět postupem obdobným, jaký je popsán v bodu 2.

Abychom si udělali názornou představu o tomto procesu, provedeme vložení bodů z tabulky 1.1 v pořadí Praha, Brno, Ostrava, Plzeň, Olomouc, Liberec, Zlín, Kladno.

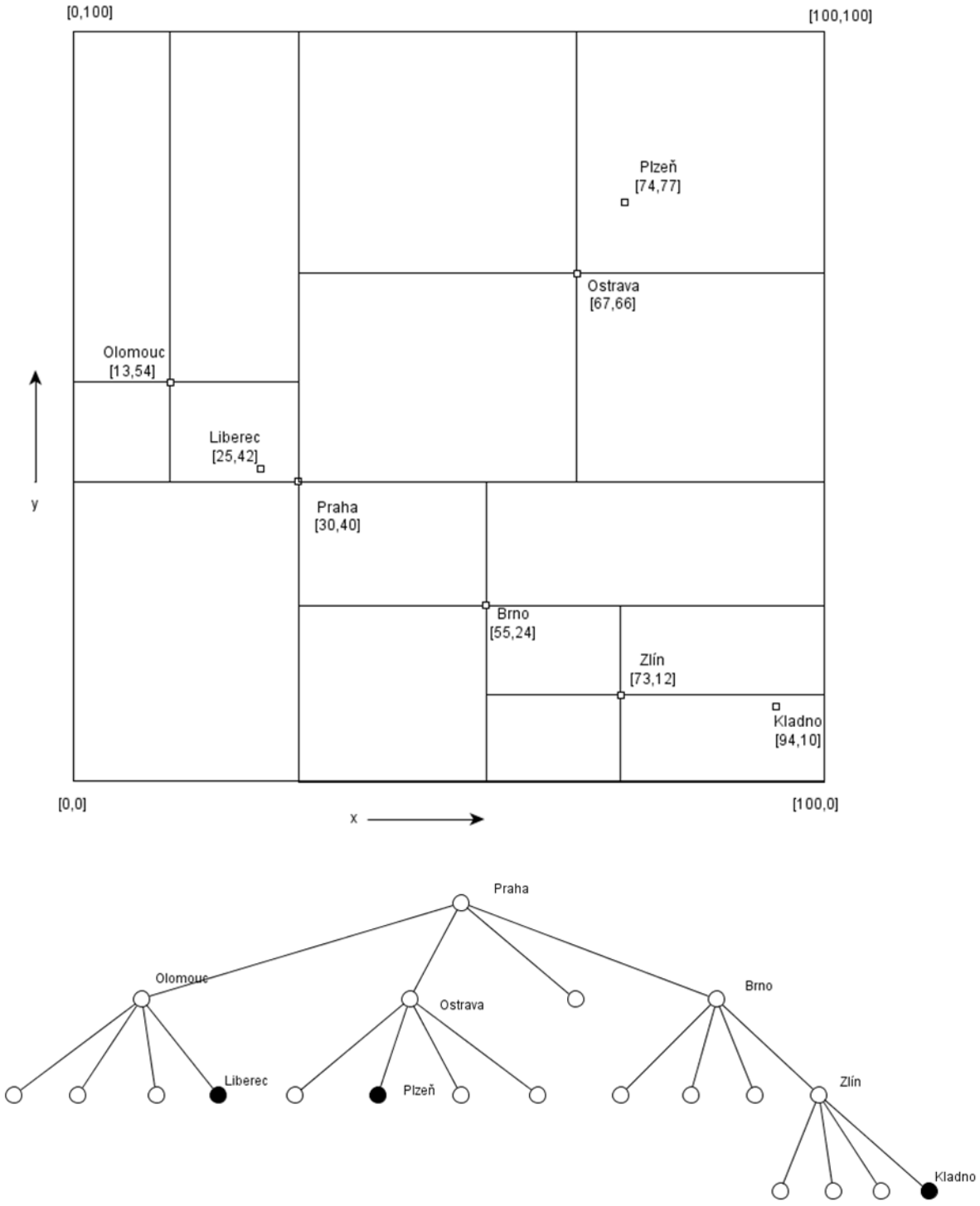
1. Nejprve tedy standardně vložíme bod Praha se souřadnicemi [30,40] do prázdného stromu, čili tento bod bude kořenem stromu.
2. Následně porovnáme bod Brno s bodem Praha. Je zřejmé, že se nenachází na stejném umístění. Určíme tedy kvadrant, ve kterém bude bod Brno ležet. Porovnáme souřadnice Prahy a Brna a zjistíme, že bod Brno bude ležet ve čtvrtém, SE kvadrantu. Podobu kvadrantového stromu v této fázi můžeme vidět na obrázku 3.2
3. Dalším vkládaným bodem je Ostrava. Ten se opět porovná s kořenem stromu, bodem Praha. Body stejné nejsou a Ostrava bude ležet ve druhém kvadrantu (NE).
4. Další je Plzeň. Opět se porovná s Prahou, souřadnice jsou různé, takže se pokračuje ve druhém kvadrantu. Tento kvadrant však není prázdný, leží zde bod Ostrava. Tudíž se Plzeň porovná s Ostravou. Body jsou odlišné, dále se pokračuje ve druhém kvadrantu. Zde je již pouze NIL uzel, takže se zde uloží bod Plzeň.
5. Jako další bod vložíme Olomouc. Ten se opět porovná s Prahou a opět body budou různé. Pokračujeme prvním kvadrantem. Zde zatím není žádný bod, takže se zde uloží bod Olomouc. Podobu kvadrantového stromu v této fázi můžeme vidět na obrázku 3.3
6. Dále vkládáme Liberec. Zase se porovná s Prahou, pokračovat se bude v prvním (NW) kvadrantu. Zde je již Olomouc, takže následuje porovnání. Body nejsou identické, pokračujeme dále ve čtvrtém (SE) kvadrantu. Zde je NIL uzel, tudíž uzel asociovaný s bodem Liberec uložíme právě zde.
7. Pokračujeme bodem Zlín. Porovnáme s Prahou, body si nejsou rovny, pokračujeme ve čtvrtém (SE) kvadrantu. Zde je uloženo Brno, následuje tedy porovnání. Body nejsou stejné, pokračujeme v uzlu, reprezentujícím čtvrtý kvadrant. Zde je neexistující uzel, takže bod Zlín umístíme zde.
8. Poslední bod, který si na zkoušku vložíme, je Kladno. Zde je identický postup jako v bodu 8, nakonec však ještě porovnáme se Zlínem a jelikož body nejsou stejné, Kladno umístíme místo NIL uzlu, reprezentujícím čtvrtý kvadrant. Výsledný strom i dekompozice prostoru je demonstrována na obrázku 3.4.



Kvadrantový strom a dekompozice prostoru po vložení bodů Praha a Brno 3.2



Kvadrantový strom a dekompozice prostoru po vložení bodů Praha, Brno, Ostrava, Plzeň a Olomouc 3.3



Výsledný kvadrantový strom s dekompozicí prostoru 3.4

Vidíme tedy, že díky jedinému bodu Kladno vzrostla hloubka stromu o 1. Z obrázku je taktéž patrné, že strom je relativně vyvážený, není zde příliš mnoho bodů v jednom kvadrantu, body jsou taktéž poměrně rovnoměrně rozmístěny po prostoru. Také je zde patrný vliv pořadí vkládání bodů do

stromu. Vezmeme si příklad, kdy jako první bod vložíme Olomouc a dále bychom pokračovali body Ostrava, Plzeň, Liberec, Praha, Brno, Zlín, Kladno. Je snadno odhadnutelné, že hloubka stromu by byla 5, což je o dvě úrovně víc než v našem původním příkladu.

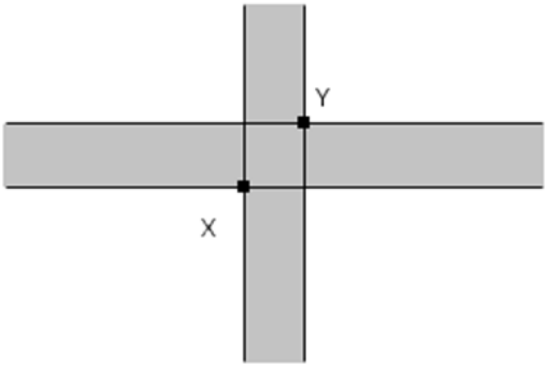
Je také možné, že nastane případ, kdy vkládáme bod, který leží na rozhraní dvou kvadrantů. V takovém případě použijeme pravidlo, popsané u metody fixed grid [1]. Toto pravidlo říká, že každý kvadrant je otevřen shora a zprava a uzavřen zdola a zleva. Pokusíme se například vložit bod Třinec se souřadnicemi [55,30]. Nejprve standardně porovnáme s Prahou, body se nerovnají, pokračujeme ve čtvrtém (SE) kvadrantu. Zde je umístěn bod Brno, opět porovnáme a opět nejsou body stejné. Nicméně souřadnice *x* se rovnají. Z naší konvence využijeme pravidlo, že levá strana kvadrantu je uzavřená. Z toho vyplývá, že bod Třinec leží ve druhém (NE) kvadrantu.

Z popsaného principu je zřejmé, že celkový objem práce nutné pro vybudování kvadrantového stromu je roven počtu kroků k nalezení všech prvků stromu. Tato práce se pohybuje pro *N* bodů okolo hodnoty $N * \log_4 N$, čili průměrná složitost vložení bodového prvku je $O(\log_4 N)$ [3]. Očekávaná hloubka stromu se pohybuje okolo hodnoty dané vztahem $\frac{(2 * c)}{(d * \ln N)}$, kde c je konstanta nabývající hodnoty 4,31107, *N* počet vložených bodů a *d* dimenze kvadrantového stromu [10,11]. Pro náš případ je očekávaná hloubka 2,07, nicméně hloubka stromu v našem příkladu je 3. Je to způsobeno tím, že body Brno, Zlín a Kladno leží téměř na diagonální přímce. Tento případ nerovnoměrného rozložení bodů v prostoru byl popsán v části 3.1.

3.1.2 Smazání prvku z kvadrantového stromu

Pro odstranění prvku z kvadrantového stromu existuje několik možností. Nejjednodušší je zřejmě ta, kterou navrhnuli Raphael Finkel a Jon Bentley [3]. Tato metoda spočívá v tom, že se odstraní daný uzel a všechny prvku podstromu s kořenem v daném odstraňovaném uzlu se vloží do stromu znova. Pokud tedy v našem příkladu na obrázku 3.4 budeme chtít smazat bod Brno, budeme po jeho smazání nuceni vložit do stromu body Zlín a Kladno. V tomto případě je to sice přijatelná metoda, nicméně u obsáhlých stromů není nejefektivnější, protože čím blíž bude odstraňovaný uzel kořeni stromu, tím více bodů bude třeba znovu vložit do stromu. Tato metoda je tedy vhodná pouze v případě, že chceme odstranit list stromu, případně uzel, který má jako všechny potomky listy stromu.

O něco sofistikovanější metoda byla vyvinuta profesorem Hananem Sametem [12]. Zkráceně řečeno, daný uzel se vymaže a nahradí se „nejbližším“ uzlem. Daný uzel se musí vybrat tak, aby jeho „hatched“ region byl co nejmenší (neměl by se v něm nacházet jiný bod uložený ve stromu). Na obrázku 3.5 vidíme, jak takový hatched region vypadá.



Hatched region mezi uzly X a Y. 3.5

Nalezení uzlu, který by v hatched regionu neměl žádné jiné uzly je nicméně ve valné většině případů pracné, navíc s nejistým výsledkem. Takový bod nemusí totiž vůbec existovat.

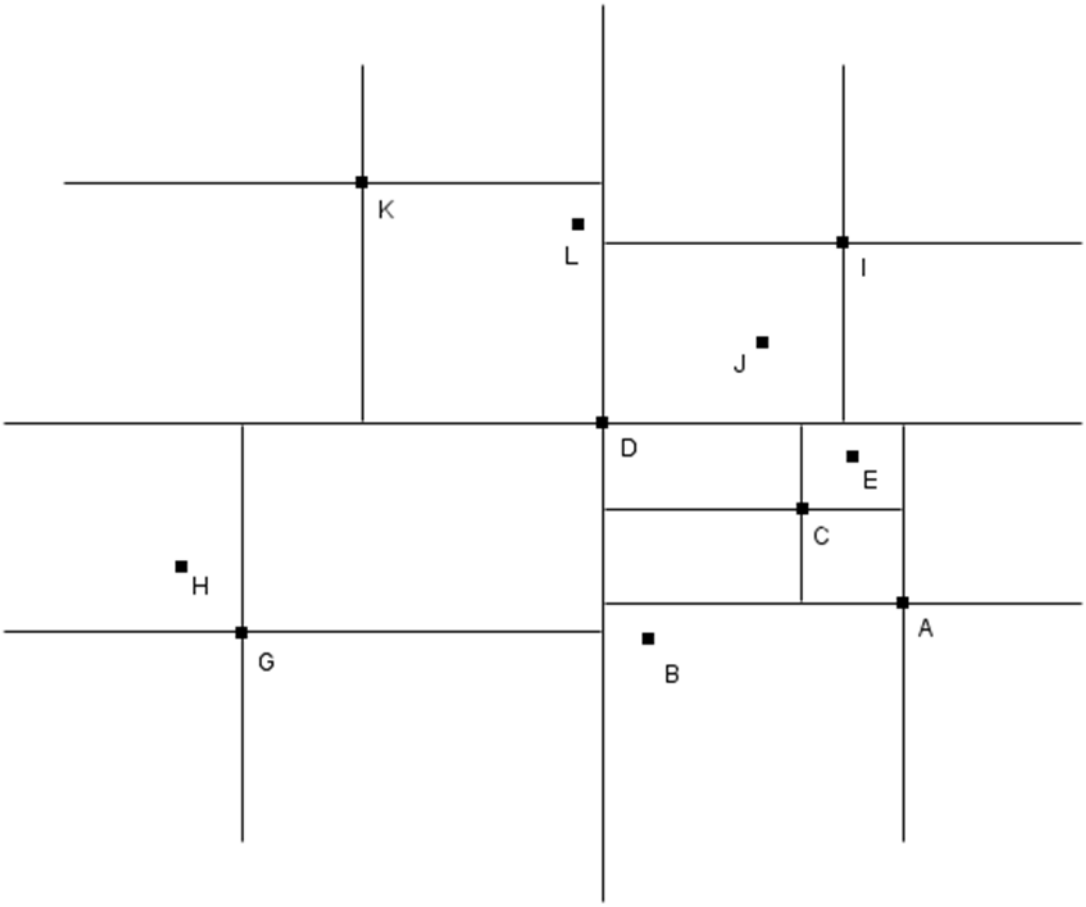
Postup vyhledání nejbližšího uzlu spočívá v podobném principu jako u binárních vyhledávacích stromů, nicméně jsou zde jistá specifika. Zatímco u binárních vyhledávacích stromů můžeme s úspěchem najít nejbližší uzel (nejpravější uzel levého podstromu, případně nejlevější uzel pravého podstromu), u kvadrantových není zřejmé, který uzel je nejbližší současně souřadnici *x* a *y* mazaného bodu.

Algoritmus mazání spočívá v nalezení čtyř kandidátských uzlů (candidate nodes) [1], jeden pro každý kvadrant vzhledem k mazanému bodu. Vyhledávání probíhá v opačném kvadrantu.

Pro ukázku vidíme na obrázku 3.6 kvadrantový strom, přičemž chceme smazat bod *D*. Ukážeme si nalezení kandidátského uzlu pro SE kvadrant.

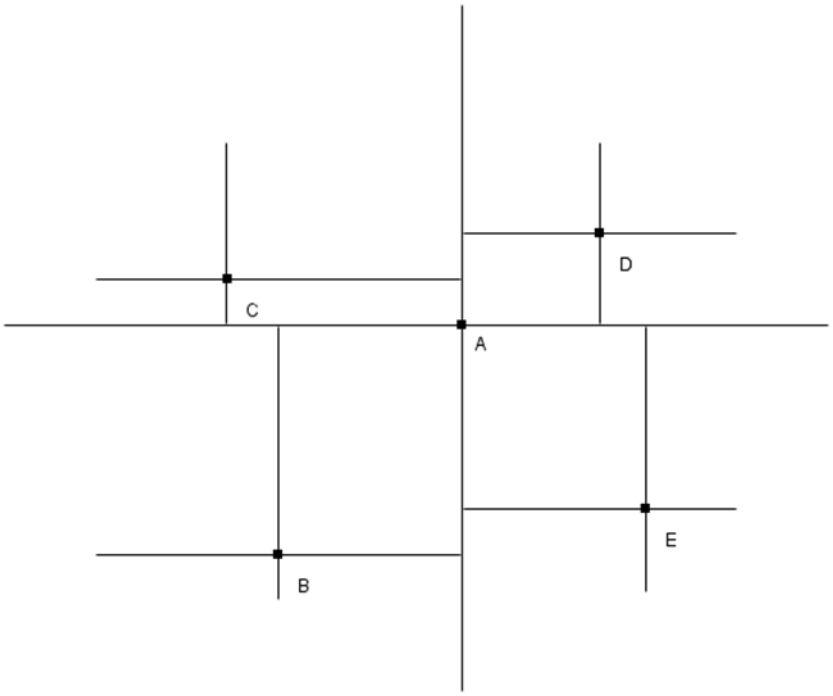
1. Vyhledáme potomka uzlu *D* pro SE kvadrant. Tímto potomkem je bod *A*.
2. V následujícím vyhledávání ve stromu budeme procházet opačný kvadrant, čili NW kvadrant. Vyhledáme tedy potomka uzlu *A* nacházejícího se v NW kvadrantu. Tímto bodem je bod *C*.
3. Bod *C* už nemá žádné další potomky, kteří by se nacházeli v NW kvadrantu. Bod *C* se tedy stává kandidátským uzlem.

Pro ostatní kvadranty bychom pokračovali identickým způsobem. Jako kandidátské uzly bychom nalezli body *G*, *L* a *J*.

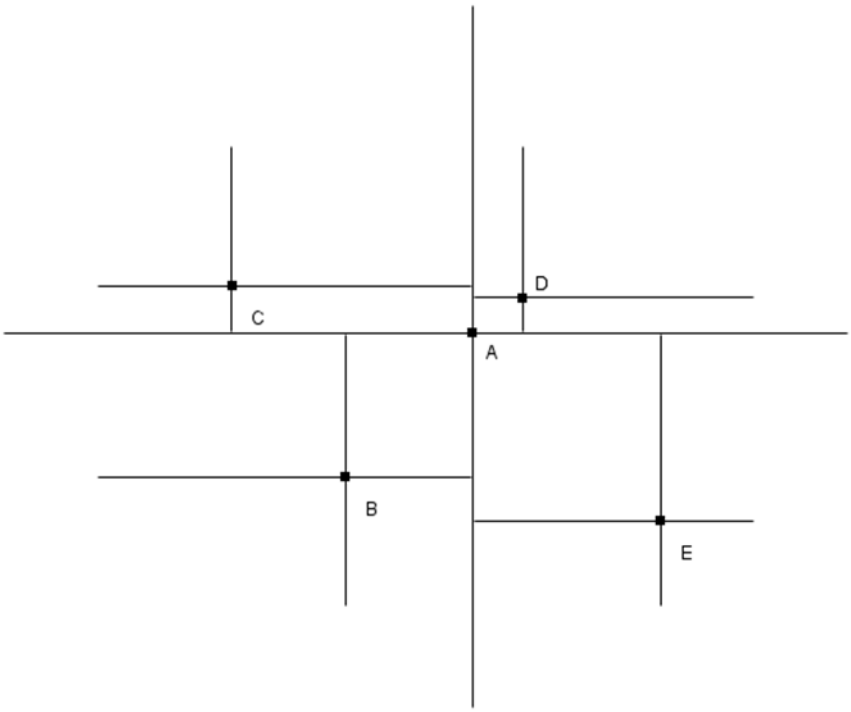


Ilustrační příklad pro mazání prvku z kvadrantového stromu 3.6

V případě, že jsme nalezli množinu čtyř kandidátských uzlů, se pokusíme z těchto uzlů najít nejvhodnější pro nahrazení mazaného uzlu. Jak již bylo řečeno, tento uzel by měl mít pokud možno prázdný (nebo alespoň co nejmenší) hatched region. Kritéria určující, který uzel je nejideálnější, jsou dvě. Kritérium 1 stanovuje bod nejbližší k oběma osám, které protínají mazaný uzel. Je zřejmé, že ne vždy lze pomocí tohoto postupu nalézt pouze jediný bod, v některých případech nenalezneme žádný uzel, v jiném zase můžeme nalézt více vhodných uzlů, respektive uzlů, splňující kritérium 1. Na obrázcích 3.7 a 3.8 můžeme vidět ukázky, kdy kritérium 1 selhalo a je nutné přistoupit ke kritériu 2. Kritérium 2 vybere z daných uzlů právě ten uzel, který má nejmenší metrickou vzdálenost k odstraňovanému uzlu. Ke kritériu 2 se uchylujeme až poté, co selhalo kritérium 1.

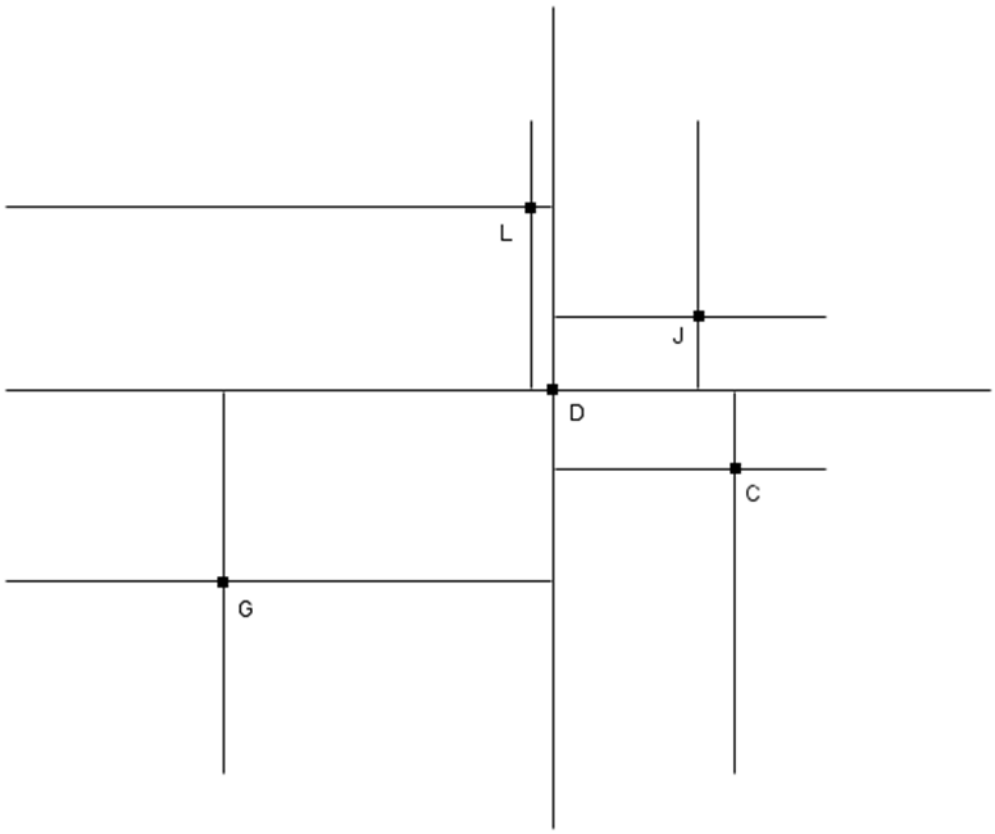


V tomto kvadrantovém stromu bychom pomocí kritéria 1 nenalezli žádný vhodný uzel 3.7



V tomto případě kritérium 1 splňují dva body a to bod B a bod D 3.8

Vraťme se k našemu ukázkovému příkladu. Pro ilustraci nalezené kandidátské uzly uvádím na obrázku 3.9.



Nalezení kandidáti v ilustračním příkladu 3.9

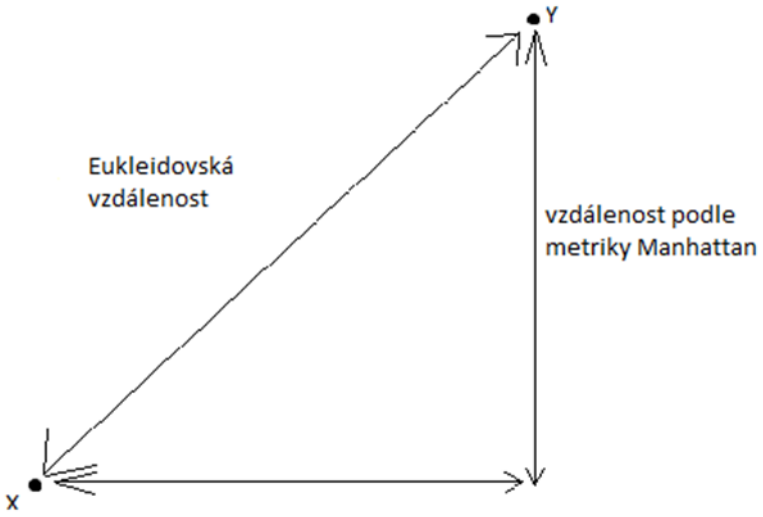
Pokusíme se určit uzel, který splňuje kritérium 1. Pro každý kandidátský uzel provedeme porovnání s jeho dvěma sousedy, čili body v sousedícím kvadrantu.

- Bod *L*:
  - s bodem *G* - porovnáváme vzhledem k svislé ose, bod *L* je blíž k této ose než *G*.
  - s bodem *J* – porovnáváme vzhledem k vodorovné ose, zde je blíže bod *J*.
  - vyhodnocení: Bod *L* nesplňuje kritérium 1.
- Bod *J*:
  - s bodem *C* – porovnáváme vzhledem k svislé ose, bod *J* je blíže
  - s bodem *L* – porovnáváme vzhledem k vodorovné ose, bod *J* je blíže
  - vyhodnocení: Bod *J* splňuje kritérium 1.

Tímto způsobem bychom pokračovali s body *C* a *G*. Došli bychom k závěru, že oba tyto body nesplňují kritérium 1. V tomto případě bychom tedy nemuseli přejít ke kritériu 2, jelikož jsme našli

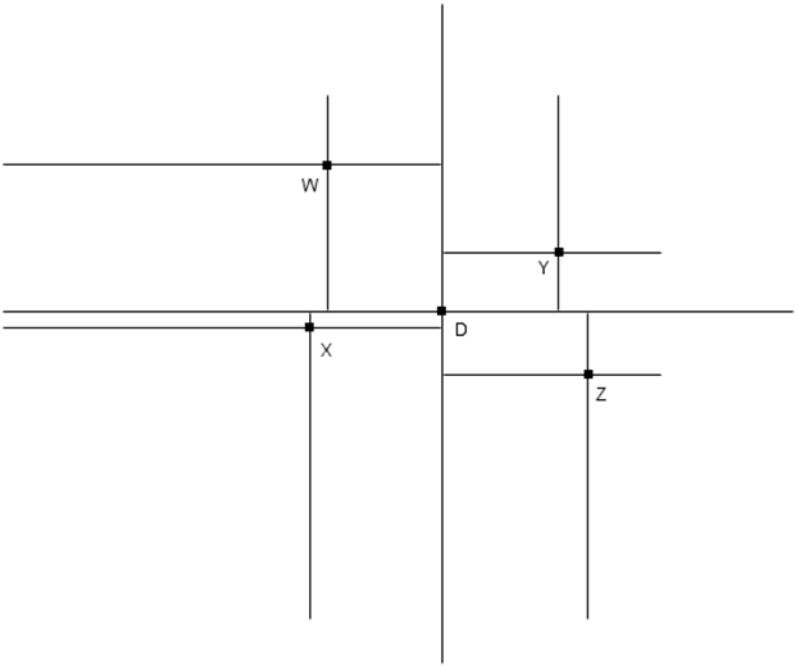
právě jeden bod, který splňuje kritérium 1 a to je bod *J*. Tímto bodem bychom tedy nahradili mazaný bod *D*.

Pokud však nastane případ, kdy pomocí kritéria 1 nenalezneme žádný, nebo naopak více vhodných bodů, musíme přejít ke kritériu 2, které spočívá ve vybrání bodu s nejmenší metrickou hodnotou. K vyjádření této hodnoty se namísto Eukleidovské vzdálenosti dvou bodů používá metrika Manhattan, což je součet vzdáleností k osám, procházejících danými body [1]. Můžeme si ji představit, jakoby jsme měřili vzdálenost po vodorovné a svislé ose, místo po diagonále.



Rozdíly v interpretaci Metrické a Euklidovské vzdálenosti 3.10

Samotné kritérium 2 však nezaručí, že v hatched regionu mezi bodem, splňujícím kritérium 2 a odstraňovaným bodem nebude žádný jiný kandidátský klíč. Na obrázku 3.11 je odstraňovaným bodem bod *D*, kritérium 2 splňuje bod *X*, zatímco bod *Y* vyhovuje kritériu 1. V hatched regionu bodu *X* se však nachází bod *W*, přičemž v hatched regionu bodu *Y* se nenachází žádný jiný bod. Z tohoto důvodu je nutné nejprve nalézt vhodné kandidáty pomocí kritéria 1 a až poté využít kritéria 2 v případě, že kritérium 1 nenajde právě jednoho vhodného kandidáta. Pokud bychom použili pouze kritérium 2, je zaručeno pouze to, že v daném hatched regionu nebude více než 1 kandidátský uzel.



Ukázka nejednoznačnosti kritéria 2 3.11

Po vyhledání nejvhodnějšího kandidáta nyní můžeme provést vymazání uzlu a jeho nahrazení tímto kandidátem. Musíme však vzít v potaz situaci, kdy v hatched regionu mezi mazaným a nahrazujícím uzlem budou jiné body. Po nahrazení odstraňovaného uzlu nahrazujícím se musí tyto uzly do stromu znovu vložit. Problém však může nastat s určováním, které uzly budeme znovu vkládat. Pokud by nám nezáleželo na čase, vložili bychom všechny uzly z podstromu odstraňovaného uzlu. Tímto bychom se však připravili o rychlost pramenící z této sofistikované metody a dostali bychom se někde k rychlosti původní metody Raphaela Finkela a Jona Bentleyho. Proto se snažíme využít vlastností prostoru, které nám dovolí snížit počet uzlů, které musíme do stromu znovu vložit. Tento postup se skládá ze dvou procedur, které si pracovně pojmenujeme SKVADRANT a NOVYKOREN [1].

Představme si následující situaci:

- bod *X*, který chceme vymazat.
- kvadrant *K*, který je NE kvadrantem uzlu *X*.
- bod *Y*, ležící v kvadrantu *K* a který je kandidátem pro nahrazení

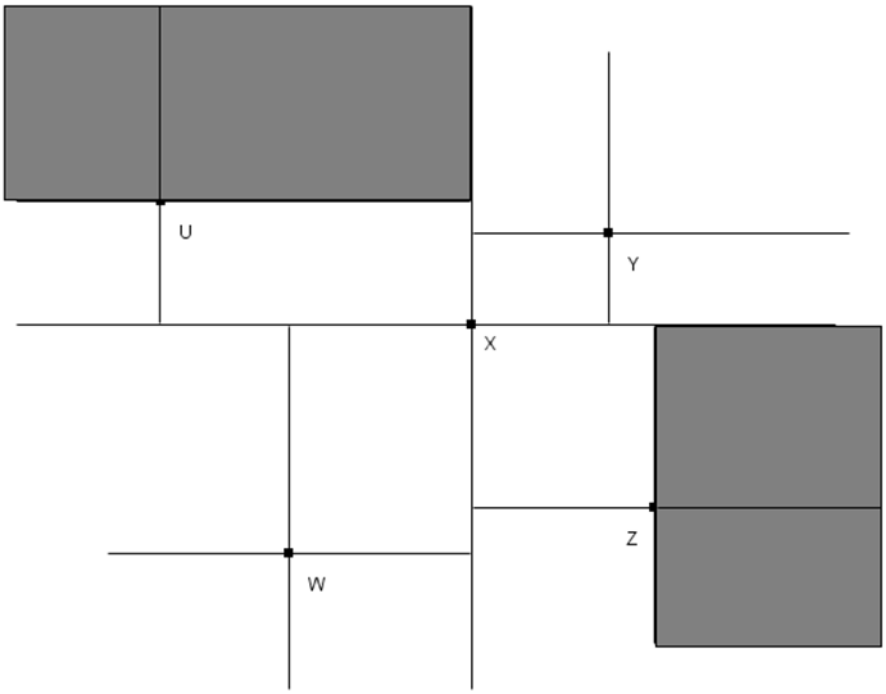
Vlastní procedura smazání uzlu *X* a jeho nahrazení bodem *Y* pracuje následujícím způsobem:

1. Použití procedury SKVADRANT na oba kvadranty sousedící s kvadrantem *K*.
2. Použití procedury NOVYKOREN na kvadrant *K*.

U bodu 1 si můžeme uvědomit, že v daném případě se nahrazení bodu *X* bodem *Y* nedotkne žádného bodu v kvadrantu, který je opačný ke kvadrantu *K*, resp. nebude třeba jeho znovu vložení do stromu.

Procedura SKVADRANT pracuje následovně. Nejprve zkontroluje, zda kořen kvadrantu, na který je funkce volána, se nachází v hatched regionu mezi body *X* a *Y*.

- Pokud se nenachází v hatched regionu, dva subkvadranty tohoto kvadrantu není třeba nijakým způsobem zpracovávat, čili body ležící v těchto subkvadrantech nebude třeba vkládat do stromu znovu.
- Pokud se nachází v hatched regionu, bude třeba znovu vložit všechny body nacházející se v tomto kvadrantu a to do podstromu s kořenem v uzlu *Y*.

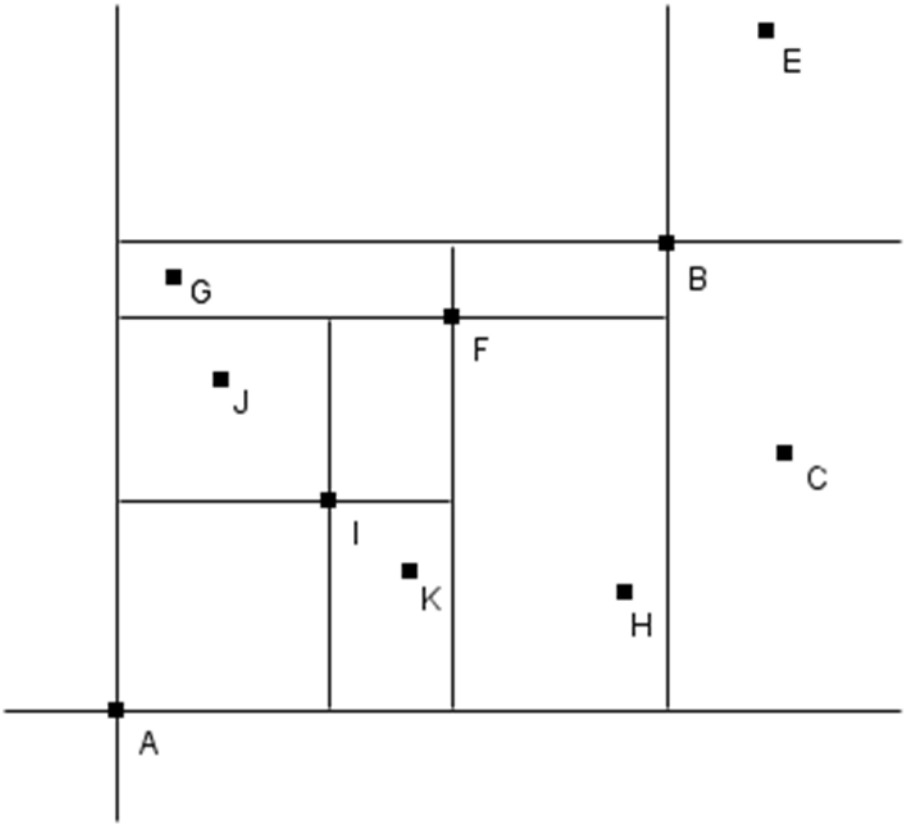


Ukázka procedury SVAKDRANT 3.12

Na obrázku 3.12 chceme vymazat bod *X* a nahradit ho bodem *Y*. Ihned je zřejmé, že jakéhokoliv bodu v kvadrantu s kořenem *W* se tato změna nijak nedotkne. Aplikujeme proceduru SKVADRANT na kvadranty s kořeny *U* a *Z*. Ani jeden z těchto kořenů neleží v hatched regionu mezi body *X* a *Y*. Z toho vyplývá, že body ležící ve dvou subkvadrantech obou těchto kvadrantů nebude třeba vkládat do stromu znovu. Tyto subkvadranty jsou označeny šedou barvou. Nicméně body, ležící ve zbylých subkvadrantech, bude nutné vložit do stromu znova a to do podstromu s kořenem v bodě *Y*.

Nyní, pokud jsme oba sousední kvadranty zpracovali procedurou SKVADRANT, zpracujeme kvadrant *K* procedurou NOVYKOREN.

První, co procedura NOVYKOREN vykoná, je nalezení kořenu kvadrantu *K* (řekněme uzel U_1). NE kvadrant tohoto uzlu zůstane nezměněn. Na NW a SE kvadranty nalezeného kořenu se zavolá již známá metoda SKVADRANT. Dále se vyhledá potomek uzlu U_1, ležící v jeho SW kvadrantu (řekněme U_2) a na jeho NW a SE kvadrant se zavolá opět metoda SKVADRANT. Takto se rekurzivně pokračuje, dokud se nenalezne uzel, který již nemá v SE kvadrantu žádné další potomky.



Ukázka procedury NOVYKOREN 3.13

Na obrázku 3.13 je znázorněn kvadrantový strom, ve kterém chceme smazat bod *A* a nahradit jej bodem *I*.

1. Jako první se zavolá metoda NOVYKOREN na NE kvadrant bodu *A*. Kořenem tohoto NE kvadrantu je bod *B*.
2. Ihned vidíme, že bod *E* zůstane nezměněn.
3. Na NW a SE kvadrant bodu *B* zavoláme metody SKVADRANT. Bod *C* se nenachází v hatched regionu mezi body *A* a *I* a tudíž jej nebude třeba do stromu znovu vkládat.
4. Vyhledáme SW potomka uzlu *B*, čili uzel *F*. Aplikujeme metody SKVADRANT na NW a SE kvadrant bodu *F*. Bod *G* se nachází v hatched regionu mezi body *A* a *I* a bude ho třeba znovu vložit do NW kvadrantu bodu *I*, poté co uzel *I* nahradí uzel *A*. Uzel *H* se

taktéž nachází v daném hatched regionu a bude ho třeba znovu vložit do SE kvadrantu uzlu *I*. Pokud by se v NE kvadrantu bodu *F* nacházel nějaký uzel či podstrom, zůstane nezměněn.

5. Opět vyhledáme SW potomka uzlu *F*, kterým je uzel *I*. Body *J* a *K* se nacházejí v hatched regionu mezi *A* a *I* a tudíž je bude nutné vložit stromu. Pokud by se v NE kvadrantu uzlu *I* nacházel nějaký uzel či podstrom, bylo by jej třeba přesunout na potomka uzlu *F*.

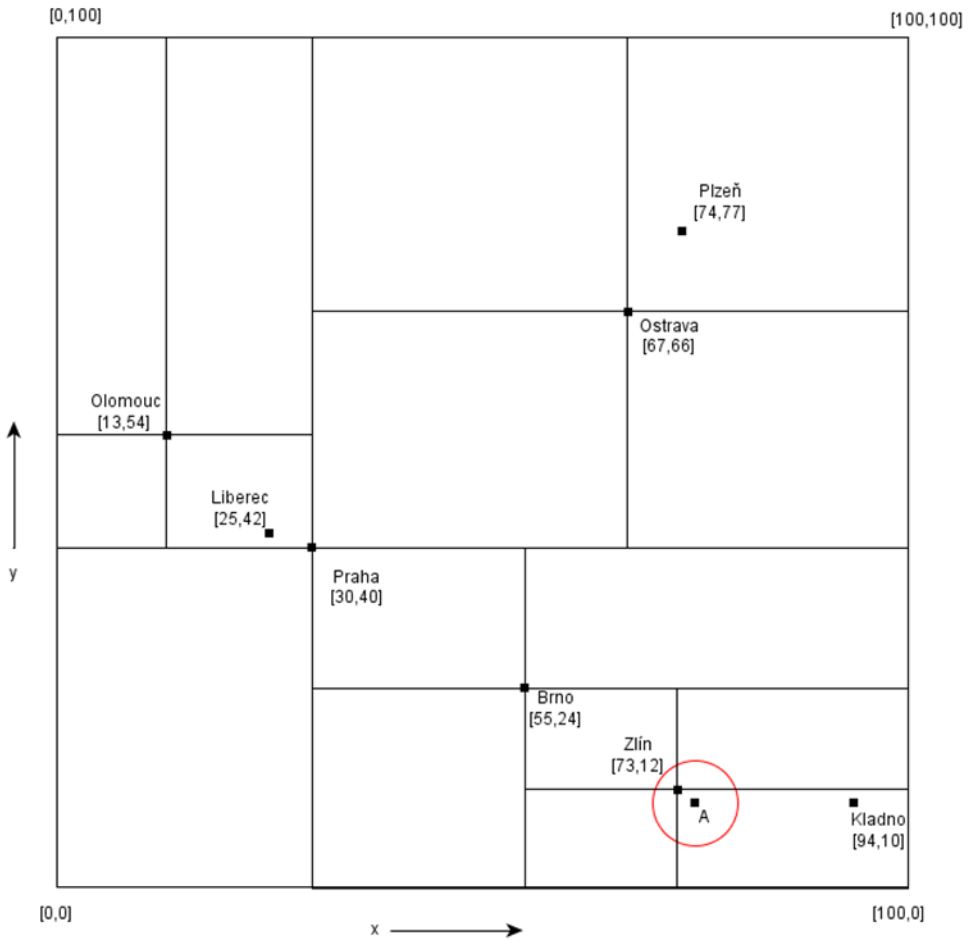
Hlavním důvodem použití této metody mazání uzlu je snížení počtu uzlů, které by bylo třeba do stromu znovu vložit. V průměrném případě se sníží tento počet uzlů o 83%, pokud nahrazující uzel splňuje kritéria 1 a 2. Pokud ze čtyř kandidátských uzlů náhodně vybereme jeden nahrazující, klesne snížení počtu uzlů, které je třeba do stromu znovu vložit na 67%. Musíme však vzít do úvahy, že se ušetří výpočetní čas, potřebný k vybrání vhodného uzlu z kandidátských uzlů [12].

3.1.3 Vyhledávání v kvadrantovém stromu

Bodové i prefixové kvadrantové stromy jsou velmi vhodné pro vyhledávání prvků v prostoru, který je dán určitým rádiem od nějakého bodu, případně plochou [1]. Ve skutečnosti jsou rozsahové dotazy nejčastějším typem dotazů u těchto datových struktur [2]. Typický rozsahový dotaz může být formulován následovně:

Nechť uzel *A* má souřadnice 75,10. Nalezněte všechny body, nacházející se ve vzdálenosti menší než 5 kilometrů od tohoto bodu.

Tento příklad si můžeme ukázat na obrázku 3.14, který vychází z ukázkového příkladu na obrázku 3.4. Hledaná oblast je vyznačena červeně. Je zřejmé, že se celá plocha hledané oblasti nachází v SE kvadrantu kořenu stromu, bodu Praha. Nebude tedy nutné prohledávat žádné body v SW, NW, NE kvadrantech uzlu Praha. Nyní se tedy přesuneme do SE kvadrantu, jehož kořenem je uzel Brno. I zde vidíme, že není nutné prohledávat SW, NW, NE kvadranty. Opět se přesuneme do SE kvadrantu, kde je kořenem uzel Zlín. Nyní již musíme prohledat všechny potomky tohoto uzlu, včetně samotného uzlu Zlín a ověřit zda leží v požadované ploše. V tomto konkrétním příkladu se jako výsledek vrátí pouze bod Zlín.



Rozsahové vyhledávání v kvadrantovém stromu 3.14

Z tohoto postupu vyplývá, že se významně sníží počet uzlů, které musíme ověřovat. Pro bodový kvadrantový strom obecně platí, že složitost vyhledávání je dána vztahem $O(d * N^{\frac{1}{(\frac{1}{d})}})$, kde *d* značí dimenzi stromu [1].

Podobným typem rozsahového dotazu je problém vyhledání uzlů v obdélníkové ploše libovolných rozměrů. Algoritmus pro vyhledávání při tomto typu dotazu vyvinuli Raphael Finkel a Jon Luis Bentley [3]. Na rozdíl od předchozího typu dotazu, který vyhledává v určité vzdálenosti od bodu, tento algoritmus je aplikovatelný jak na data v prostoru tak na obecná data. Jeden rozměr může například představovat výkon motoru automobilu a druhý spotřeba paliva. Dotaz pak může znít následovně: Vyhledejte všechny automobily, které mají výkon motoru mezi 80 a 100 kilowatty a spotřebu mezi 5 a 7 litry paliva na 100 kilometrů.

3.2 Prefixové kvadrantové stromy

Bodové kvadrantové stromy rozdělují prostor do oblastí, jejichž velikost je daná samotnými body, čili oblasti nemají stejnou velikost. Naproti tomu, prefixové stromy [1] dělí prostor rekurzivně na stejně velké části. Prostor dělí tak dlouho, dokud jsou části homogenní. Podmínky homogenity se u jednotlivých variant prefixových kvadrantových stromů liší. U MX kvadrantových stromů musí mít oblast buď všechny podprostory prázdné, nebo oblast obsahuje právě jeden bod. U PR kvadrantových stromů neobsahuje oblast více než 1 bod. Stejně jako u bodových kvadrantových stromů, dělí prefixové stromy prostor do čtyř kvadrantů u dimenze 2. Jelikož dělí prostor do stejně velkých částí, není třeba u nich uchovávat souřadnice kořenů podstromů, jelikož je lze odvodit z cesty od kořene stromu a aktuální hloubky. Taktéž zachovávají vlastnost bodových kvadrantových stromů, která spojuje menší prázdné oblasti do větších oblastí.

3.2.1 MX kvadrantové stromy

Jak již bylo řečeno, MX kvadrantové stromy patří mezi prefixové kvadrantové stromy. Můžeme si je představit, jako čtvercovou matici, kde indexované body reprezentují nenulové prvky matice [1]. Odtud také vychází název MX – matrix, čili anglicky matice. Z vlastnosti dělení prostoru na stejně velké oblasti plyne fakt, že u MX kvadrantových stromů nezáleží výsledná struktura stromu na pořadí vkládání bodů do stromu. Předpokladem pro použití MX kvadrantových stromů je diskrétnost domény souřadnic bodů.

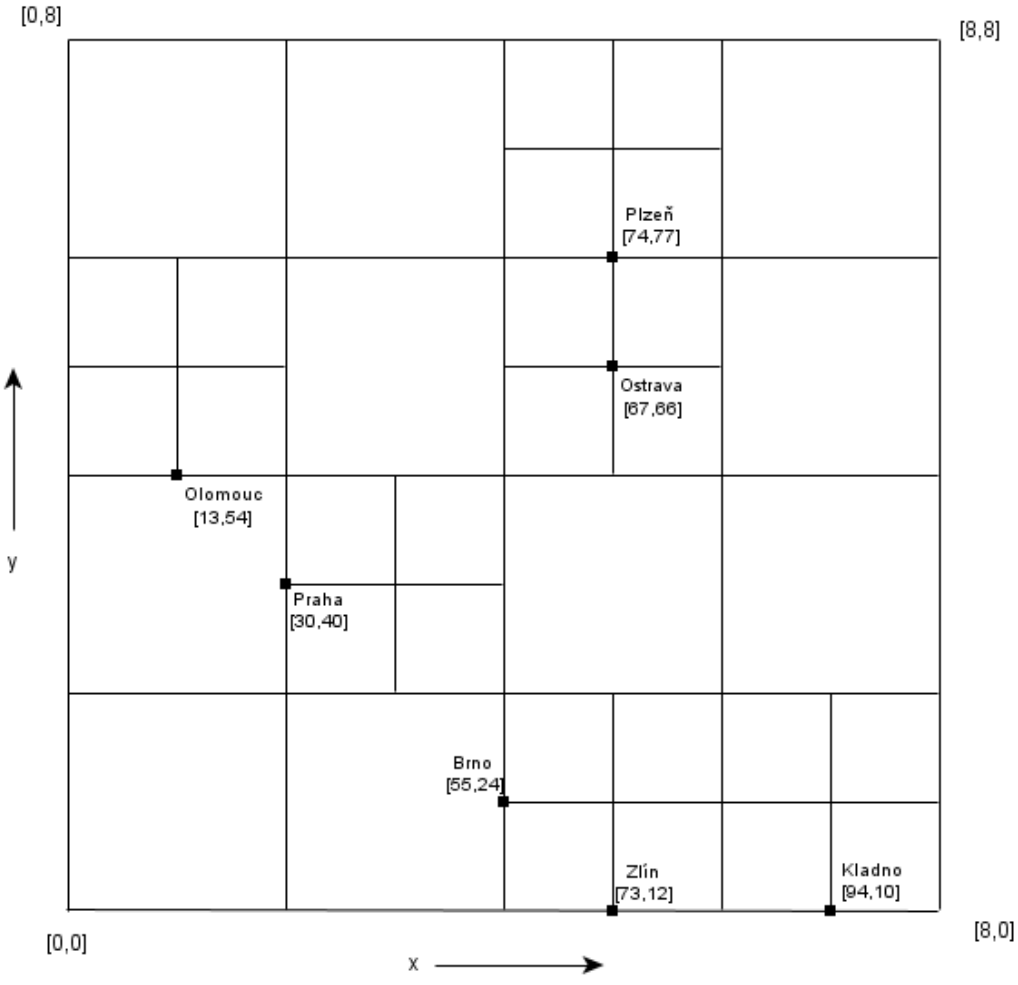
V MX kvadrantových stromech jsou všechny body uloženy v listových uzlech v největší hloubce stromu. Body reprezentují oblasti o velikosti 1×1. Mimo bodů mohou být jako listové uzly také NIL uzly, které představují prázdné oblasti, jejichž velikost může být libovolná. Čili strom o hloubce n si můžeme představit jako prostor o $2^n \times 2^n$, kde je každá oblast buď prázdná, nebo obsahuje bod.

Při konstrukci MX kvadrantového stromu je nutné nejdříve zvolit maximální hloubku stromu, ze které vyplývá také jeho rozlišovací schopnost. Pro ukázku si zvolíme jako data body z tabulky 3.1. Pro jasnější demonstraci zvolíme maximální hloubku stromu rovnu 3. Je zřejmé, že se jedná o velmi hrubé rozlišení, jelikož celý prostor stromu se bude skládat maximálně z 64 oblastí (prostor o velikosti $2^3 \times 2^3$ oblastí), ale pro takto malý počet bodů bude rozlišení dostačující. Na x-ové souřadnice bodů se aplikují mapovací funkce ve tvaru $f(x) = \frac{x}{12{,}5}$. Obdobně pro souřadnice y. Číslo 12,5 je poměr mezi velikostí původního prostoru (100) a prostoru, indexovaným MX kvadrantovým stromem (8). Každou souřadnici tedy musíme zmenšit 12,5 krát. Samotné vkládání uzlů do stromu je podobné, jako u bodových kvadrantových stromů s tím rozdílem, že body se vkládají až na úrovni 3. Porovnávají se souřadnice kořenů podstromů s vkládaným bodem. Podle výsledku porovnání se určí, kterým kvadrantem se bude pokračovat. Pokud narazíme na NIL uzel, tak oblast, reprezentovaná tímto uzlem,

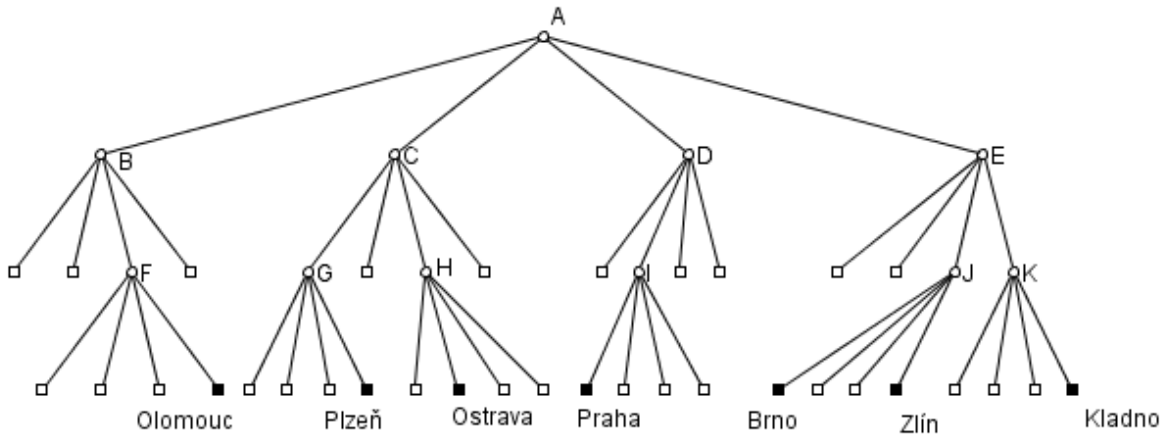
se musí rozdělit do čtyř subkvadrantů, ve kterých dále probíhá porovnávání. Pokud narazíme na NIL uzel v úrovni 3, nahradíme tento NIL uzel vkládaným bodem. Pokud na úrovni 3 narazíme na jiný bod, přepíšeme daný bod vkládaným bodem. Na obrázku 3.15 vidíme konečnou podobu prostoru a na obrázku 3.16 odpovídající strukturu MX kvadrantového stromu. Pro zobrazení je použita notace, kdy bod je asociován s levým dolním rohem oblasti.

Název	*X*	*Y*
Praha	*30*	*40*
Brno	*55*	*24*
Ostrava	*67*	*66*
Plzeň	*74*	*77*
Olomouc	*13*	*54*
Zlín	*73*	*12*
Kladno	*94*	*10*

Zdrojová data 3.1



Výsledná dekompozice prostoru pro data z tabulky 3.1 3.15



Výsledný MX kvadrantový strom pro data z tabulky 3.1 3.16

Odstraňování uzlu je u MX kvadrantových stromů mnohem jednodušší než u bodových kvadrantových stromů, jelikož všechny uzly jsou uloženy v maximální hloubce stromu, v našem případě na úrovni 3. Nemusíme totiž poté měnit uspořádání struktury stromu, stačí vyhledat daný uzel a nahradit jej NIL uzlem. Pokud však všechny listové uzly daného kvadrantu jsou NIL uzly, musíme navíc provést sloučení těchto uzlů. To provedeme tak, že vnitřní uzel, reprezentovaný předkem mazaného uzlu, nahradíme NIL uzlem. Pokud jsou na úrovni předka opět všechny uzly NIL, pokračujeme stejným způsobem, dokud není po nahrazení NIL uzlem v dané úrovni alespoň jeden vnitřní uzel.

Pro ukázku mazání uzlu odstraníme ze stromu na obrázku 3.16 bod Praha. Po jeho nahrazení NIL uzlem ale nastává případ, kdy jsou všechny uzly v tomto kvadrantu a v této hloubce NIL uzly. Tudíž je nutné provést sloučení těchto uzlů. To je provedeno tak, že vnitřní uzel I nahradíme NIL uzlem. Nicméně, i v úrovni 2 jsou nyní všechny uzly NIL. Tudíž je opět nutné provést sloučení, čili vnitřní uzel D nahradíme NIL uzlem. V úrovni 1 už jsou další vnitřní uzly, tudíž se už slučování neprovádí.

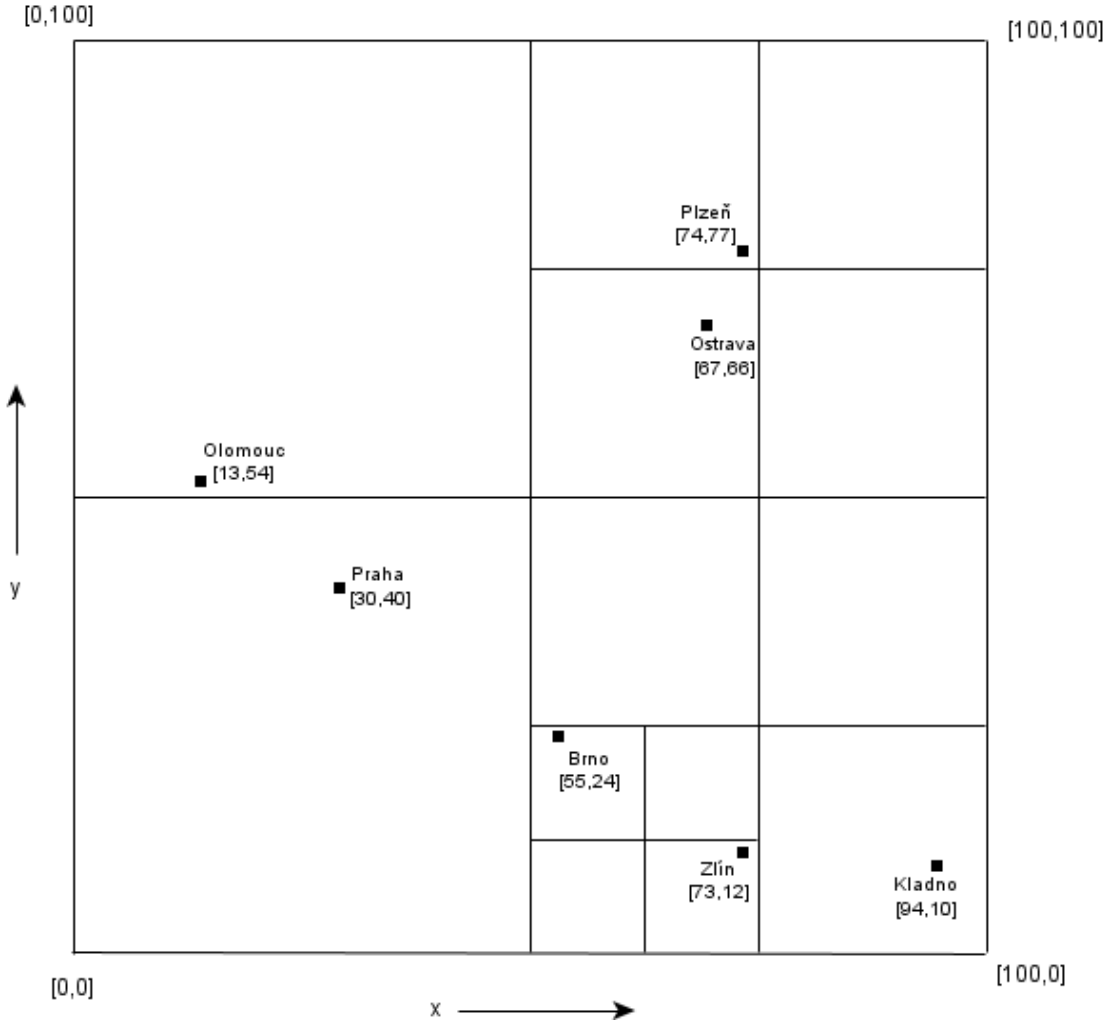
Provádění rozsahových dotazů se řídí stejnými pravidly jako u bodových kvadrantových stromů. Složitost těchto dotazů je v nejhorším případě $O(F + 2^n)$, kde n je maximální hloubka stromu a F je počet vrácených uzlů [1].

3.2.2 PR kvadrantové stromy

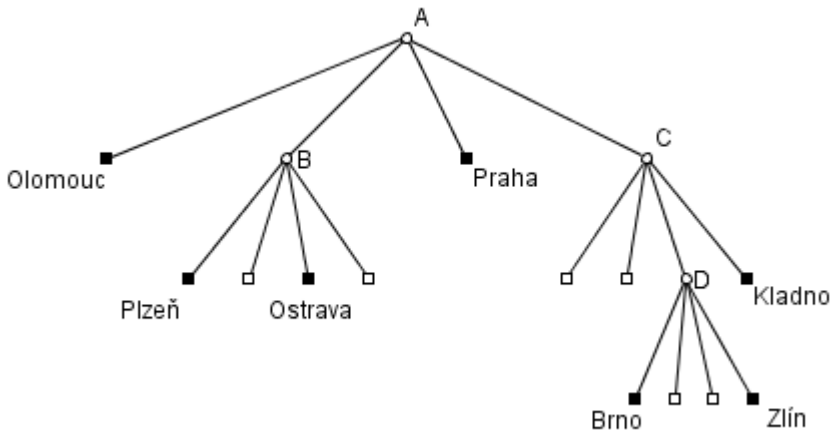
Pokud je potřeba indexovat prostor s body, jejichž doména není diskrétní, není možné použít MX kvadrantové stromy. V tomto případě je třeba využít PR kvadrantové stromy [1]. Ty taktéž patří do prefixových kvadrantových stromů a liší se od nich v tom, že body nejsou uloženy jen v největší hloubce stromu. Také zde není třeba přepočítávat souřadnice. Naopak i zde není struktura výsledného

stromu závislá na pořadí vkládání bodů do stromu. Dále zde platí pravidlo, že každý vnitřní uzel obsahuje odkazy na nejméně dva potomky, které obsahují body, případně alespoň na jeden jiný vnitřní uzel [1].

Vkládání uzlu probíhá podobně jako u MX kvadrantového stromu. Rozdíl je však v tom, že pokud nalezneme oblast, ve které již leží uzel, který má rozdílné souřadnice, je třeba tuto oblast dále rozdělit. Tento případ může nastat právě díky tomu, že uzly nemusí být uloženy v největší hloubce stromu. Oblasti dále rozdělujeme až do momentu, kdy se bude daný uzel nacházet v jiném kvadrantu, než právě vkládaný uzel. Počet těchto operací roste, pokud je Eukleidovská vzdálenost mezi již vloženým uzlem a nově vkládaným uzlem velmi malá [1]. Obrázek 3.18 zobrazuje strukturu PR kvadrantového stromu pro data z tabulky 3.1



Výsledná dekompozice prostoru pro data z tabulky 3.1 3.17



Výsledný PR kvadrantový strom pro data z tabulky 3.1 3.18

Podobně jako u MX kvadrantových stromů, i zde je odstraňování uzlů mnohem jednodušší než u bodových kvadrantových stromů, protože všechny uzly jsou uloženy v listu stromu. Opět se zde nahradí odstraňovaný uzel NIL uzlem. Pokud v tomto kvadrantu není žádný jiný bod, musíme sloučit prázdné listové uzly. Pokud je v daném kvadrantu 1 a méně bodů, je třeba tento bod přesunout o úroveň výše. Po přesunu se opět provede kontrola, zda jsou na dané úrovni dva a více uzlů reprezentujících body nebo alespoň jeden vnitřní uzel. Pokud ne, opět se provede posun bodu o úroveň výše. Takto to může pokračovat, dokud nebudou splněny podmínky, případně dokud nenarazíme na kořen stromu.

Rozsahové dotazy jsou vykonávány velmi podobným způsobem jako u MX kvadrantových stromů a mají taktéž stejnou složitost $O(F + 2^n)$, kde n je maximální hloubka stromu a F je počet vrácených uzlů [1].

4 R-stromy

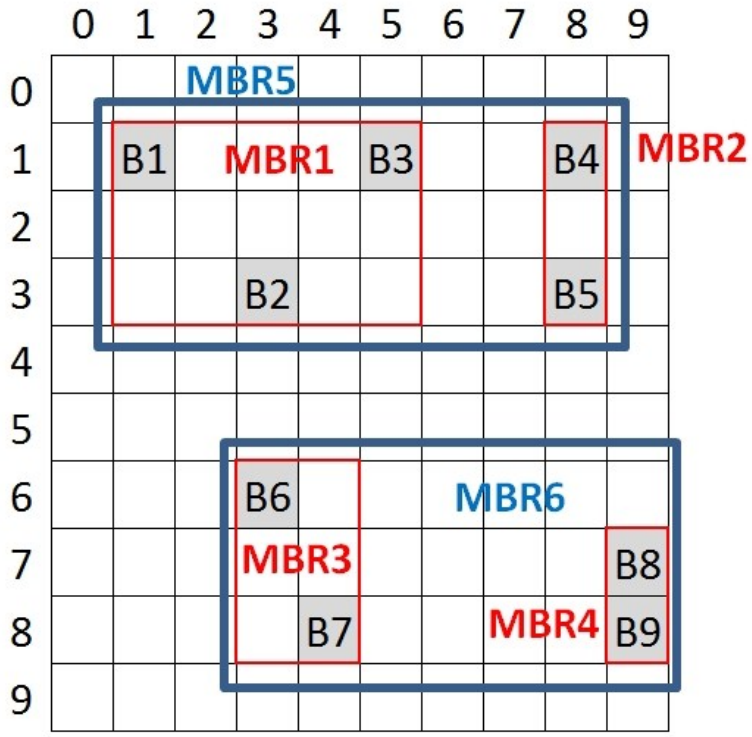
Před popisem implementace kvadrantových stromů je přiblížena problematika R-stromů, jelikož právě s touto datovou strukturou se kvadrantový strom porovnával po výkonové stránce.

Vícedimenzionální datová struktura R-strom byla navržena Antoninem Guttmanem v roce 1984 [5]. Vychází z principu B-stromu a má široké využití, od výzkumných účelů až po praktické aplikace [6]. Nejčastěji se využívá k indexování bodů v prostoru, je však také vhodná k indexování tvarů jako je obdélník nebo víceúhelník, které jsou taktéž dané body v prostoru.

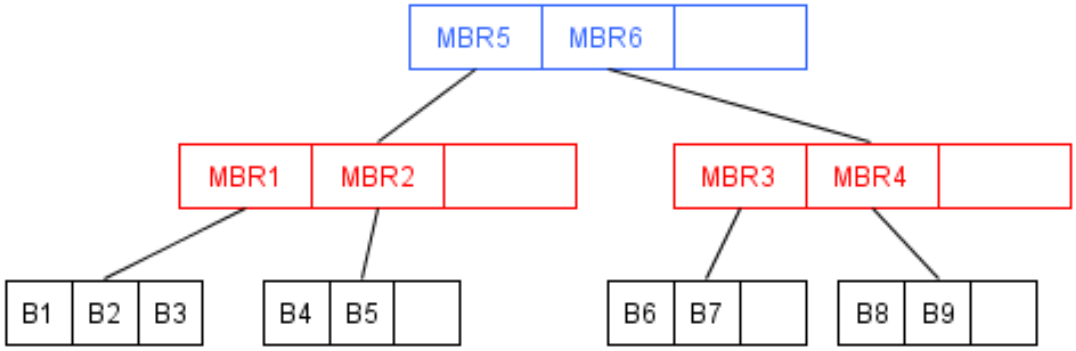
Základní myšlenka R-stromu spočívá ve sjednocování či shlukování blízkých bodů do tzv. stránek. Do stránek se shlukují i blízké minimální ohraničující obdélníky (anglicky minimum bounding rectangles, MBR). Body, které mají společný MBR jsou uloženy v listu stromu. Vnitřní uzly stromu jsou tvořeny právě hierarchií MBR. Díky tomu, že jsou data sjednocená ve stránkách, jsou R-stromy vhodné pro použití v databázových systémech.

R-strom je stejně jako B-strom vyvážený. Každá stránka má určenou svou maximální kapacitu a minimální počet záznamů.

Na obrázku 4.1 je prostor dimenze 2 s devíti body, který je indexován pomocí R-stromu. Body B1, B2, B3 jsou obsaženy v MBR1, B4 a B5 v MBR2. Minimální ohraničující obdélníky MBR1 a MBR2 jsou zase obsaženy v MBR5, který leží o úroveň výše. Obdobně pro body 6 až 9. Hierarchie minimálních ohraničujících obdélníků a indexovaných bodů je znázorněna na obrázku 4.2



Dvoudimenzionální prostor, indexovaný R-stromem 4.1



Struktura R-stromu 4.2

V případě bodového dotazu se R-strom prochází do hloubky. Nejdříve se vyhledá, ve kterém MBR leží požadovaný bod. Pokud se takový MBR nenalezne, vyhledávání skončí, bod ve stromu není. Pokud se najde, načte se relevantní potomek aktuálního uzlu. Opět se hledá MBR, ve kterém leží vyhledávaný bod. Takto se pokračuje až k listovému uzlu, kde buď leží, nebo neleží požadovaný bod. Jelikož se u R-stromu mohou MBR protínat, je možné, že existuje více cest k hledanému bodu. Všechny tyto cesty je nutné projít. Aktuální cesta je vždy ukládána do zásobníku [5].

Vkládání bodů a odstraňování uzlů je u R-stromu relativně komplikovaný problém, protože je nutné, aby strom byl vždy vyvážen (všechny datové body jsou v listových uzlech) a zároveň, aby MBR nepokrývaly příliš mnoho prázdného místa (kde nejsou žádné body). Taktéž je vhodné udržovat, aby se MBR nepřekrývaly příliš velkou plochou, jelikož je pak nutné při bodových a rozsahových dotazech projít více MBR. Pokud se tedy při vložení uzlu překročí kapacita stránky, je nutné stránku rozdělit (anglicky splitting). Existují různé algoritmy pro toto rozdělování jako například kvadratická metoda, lineární metoda, metoda Exhaustive search, R*-strom, R^+-strom.

Metoda Exhaustive search [5] například zkouší všechny možnosti, jak rozdělit daný uzel, aby výsledné uzly zachovávaly minimální a maximální počet záznamů v nově vytvořených uzlech. Z těchto možných rozdělení poté vybere to, kde MBR mají nejmenší možnou plochu.

Oproti klasickým R-stromům, které minimalizují plochu MBR, R*-stromy [7] minimalizují taktéž jejich překrývání.

R^+-stromy [8] rozdělují uzly vždy tak, aby výsledné uzly byly disjunktní. Může však nastat případ, kdy indexovaný objekt leží v obou uzlech současně. Poté se hledá dvojice MBR, které mají co nejméně takovýchto objektů, a poté na tyto objekty odkazují oba dva uzly.

5 Implementace

Následující část diplomové práce popisuje konkrétní implementaci kvadrantových stromů s podrobným popisem funkcí, algoritmů a taktéž jsou popsány problémy, které se při implementaci vyskytly.

5.1 Způsob implementace

Pro implementaci byl použit framework skupiny DBRG – Database research group. Tuto skupinu vede docent Michal Krátký, se kterým jsem na implementaci nejvíce spolupracoval. V tomto frameworku jsou již implementovány základní datové struktury, pomocné typy a taktéž několik vícedimenzionálních struktur [4]. Mým hlavním úkolem pro implementaci tedy bylo přidat a naprogramovat další třídy datové struktury kvadrantových stromů. Měl jsem možnost využít generických datových struktur, ze kterých dědí i jiné datové struktury v tomto frameworku jako je například *cPagedNode*, kde jsou implementovány vnitřní algoritmy ve smyslu přečtení návazného uzlu po předání indexu apod. [4]. Mezi další prvky, které jsem využil, patří *cTuple*, což reprezentuje n-tici vícedimenzionálních souřadnic. Podrobnější popis využitých prvků je obsažen v dalším textu.

Pro implementaci byl využit jazyk C++. Tento jazyk je použit především z důvodů větší kontroly programátora nad využitím operační paměti a velmi vysoké rychlosti aplikací napsaných v C++. Po implementaci bylo provedeno několik výkonnostních testů a výsledky byly porovnány s nejbližší datovou strukturou - R-stromem.

5.2 Vlastní implementace

5.2.1 Vytvoření struktury tříd

Prvním úkolem při implementaci bylo vytvoření základní struktury tříd. Zde bylo nutné zachytit všechny prvky, potřebné pro správnou funkčnost kvadrantových stromů v souladu s filozofií frameworku. Následuje výpis tříd a stručný popis, za jakým účelem byly vytvořeny.

1. **cQTree.h** – Základní třída celé struktury. Obsahuje elementární konstanty nezbytné pro funkčnost struktury. Mezi ně patří:

- DIMENSION – označuje dimenzi, ve které se bude pracovat
- MAX_HEIGHT – maximální výška stromu
- DEBUG_OFF – indikace ladění kódu. Tato konstanta je nastavená na 0 a značí vypnuté ladění kódu.
- DEBUG_INDIVIDUALLY - indikace ladění kódu. Tato konstanta je nastavená na 1 a značí podrobné ladění kódu. Používá se v případě, pokud je třeba testovat provádění

vkládání či hledání jednotlivých n-tic, správné počítání kvadrantů apod. Zobrazují se velmi podrobné výpisy aktuálních souřadnic uzlu a další informace

- DEBUG_BULK - indikace ladění kódu. Tato konstanta je nastavená na 2 a značí ladění kódu při hromadném vkládání, či hledání. Používá se pro výpis počtu vložených nebo vyhledaných uzlů po určitých hodnotách, například po každých 10 000 vložených uzlů.
- DEBUG – do této konstanty se přiřazuje hodnota uvedených konstant podle žádaného stupně ladění.
- DEBUG_RANGE_QUERY – velmi podobná funkce jako u konstanty DEBUG, avšak platí pouze pro metodu *RangeQuery*. Je vytvořená pro případ, kdy je nutné hromadně vložit uzly a zároveň zkoumat jednotlivé kroky v metodě *RangeQuery*, čili není možné použít konstantu DEBUG.
- NODE_INSERTED, NODE_ALREADY_INSERTED, NODE_NOT_INSERTED – konstanty, reprezentující hodnoty, které vrací metoda Insert. NODE_INSERTED je reprezentuje 0, NODE_ALREADY_INSERTED číslo 1 a NODE_NOT_INSERTED číslo 2 a tudíž značí špatné vložení uzlu
- S následujícími konstantami se pracuje v metodě *RangeQuery* ve vícerozměrném poli sloužícím pro uložení dočasných hodnot uzlů:
  - STACK_NODE_ID – číslo uzlu
  - STACK_NEXT_QUADRANT_TO_EXAMINE – indikátor, podle kterého se určí, jaký další kvadrant se má zkontrolovat
  - NO_QUADRANT – indikátor, označující stav, kdy už nezbývá prozkoumat žádný další kvadrant v současné hloubce.

Dále tato třída obsahuje základní metody pro práci se stromem:

- Create (cQTreeHeader<TKey> *header, cQuickDB* quickDB) – slouží pro vytvoření instance kvadrantového stromu v dočasné databázi v paměti. Jako parametr se metodě předá již vytvořená instance hlavičky stromu a pointer ukazující na dočasnou databázi.
- Close() – slouží k uzavření struktury a její vymazání z paměti.
- Insert (TKey &item, char* data) – slouží pro vložení uzlu do stromu. Jako parametr přijímá generický typ reprezentující souřadnice uzlu (při běhu se místo *TKey* pracuje právě s *cTuple*) a data která daný uzel bude obsahovat.
- Find(TKey &item, char* data) – slouží pro hledání uzlu ve stromu. Parametry metody jsou stejné jako u metody *Insert* a mají taktéž stejný účel.
- RangeQuery(TKey &leftUpperSearchedItem, TKey &rightLowerSearchedItem) – slouží pro vyhledání všech bodů uvnitř zadaného obdélníku (rozsahový dotaz). Jako parametry jsou předány instance třídy *cTuple* reprezentující levý horní a pravý dolní roh obdélníku.

- IsInSpace(TKey &insertedItem) – pomocná metoda sloužící pro otestování, zda je vkládaný či hledaný uzel v prostoru kvadrantového stromu. Prakticky jde o to otestovat, zda souřadnice bodu nepřekračují limit daný použitým datovým typem. Například pro unsigned int nesmí být žádná souřadnice větší než 4 294 967 295.
- IsInNode(TNode *currentNode,TKey &insertedItem) – slouží pro otestování, zda aktuální uzel je ekvivalentní hledanému uzlu. Jako parametr se předává pointer, který ukazuje na aktuální uzel a odkaz na instanci *cTuple*, ve které jsou uloženy souřadnice vkládaného nebo hledaného uzlu.
- ComputeQuadrant(TNode *currentNode, TKey &insertedItem) – tato metoda slouží pro výpočet kvadrantu, ve kterém leží vkládaný bod. Z aktuálního uzlu se vyextrahují souřadnice, které následně slouží jako střed a porovnávají se se souřadnicemi vkládaného uzlu. Následně se vrátí číslo kvadrantu, ve kterém vkládaný bod leží. Jako parametry slouží opět pointer ukazující na aktuální uzel a odkaz na instanci *cTuple*, ve které jsou uloženy souřadnice vkládaného nebo hledaného uzlu.

2. **cQTreeHeader.h** - Tato třída reprezentuje hlavičku stromu. Neobsahuje už žádné důležité konstanty. Obsahuje následující základní metody:

- Init() – V této metodě se provádí inicializace stromu. To zahrnuje kromě volání nadřazené statické metody *Init* ve třídě *cTreeHeader* nastavení názvu stromu, verze programu a verze buildu.
- Write(cStream* stream), Read(cStream* stream) – tyto metody slouží ke čtení, případně k zápisu datového proudu do paměti.
- GetSpaceDescriptor() – vrací space descriptor (popisovač prostoru) aktuální instance stromu.
- HeaderSetup(uint blockSize) – Nejdůležitější metoda této třídy. Provádí následující nastavení:
  - Nastavuje počet itemů v listovém uzlu. Itemem se myslí generický typ *TKey*, který při běhu reprezentuje *cTuple*. V našem případě se nastaví na 1.
  - Nastaví počet linků na následující uzly. V dimenzi 2 bude počet následujících uzlů 4. Obecně tedy bude počet následujících uzlů 2^d, kde *d* značí dimenzi prostoru.
  - Zavolá se metoda *ComputeNodeCapacity(blockSize)*, která je ve třídě *cQTreeNodeHeader*.

3. **cQTreeNode.h** – Tato třída reprezentuje uzel stromu. Obsahuje pouze prázdný konstruktor a destruktor.

4. **cQTreeNodeHeader.h** – Tato metoda reprezentuje hlavičku uzlu a slouží hlavně pro práci s uzlem. Obsahuje metody:

- GetSpaceDescriptor() – vrací aktuální space descriptor.

- ReadNode(cNode* block, cStream* stream), WriteNode(cNode* block, cStream* stream) – slouží pro zápis, respektive čtení uzlu z paměti.
- CopyNode(cNode* dest, cNode* source) – slouží pro kopírování jednoho uzlu do druhého. Ukazatele na oba uzly jsou předány jako parametr.
- CreateCopy(unsigned int inMemSize) – slouží pro vytvoření kopie hlavičky uzlu. Na konci vytvoření kopie se na ni volá metoda *ComputeNodeCapacity*.
- ComputeNodeCapacity(unsigned int blockSize) – nejdůležitější metoda této třídy. Slouží pro výpočet reálné velikosti uzlu. Jako parametr se předává velikost bloku na pevném disku.

Obsahuje taktéž důležité konstanty:

- capacity – uvádí počet itemů(instancí třídy *cTuple*) v uzlu.
- linkCapacity – reprezentuje počet odkazů na následující uzly.

Další část této kapitoly je věnována podrobné implementaci jednotlivých metod. K vysvětlení principu není využit jazyk C++, jelikož ten lze najít v příloze. Místo C++ je použit pochopitelnější pseudokód.

5.2.2 Třída cQTree

Jak již bylo řečeno, v této třídě jsou uložené důležité konstanty a základní metody pro práci se stromem, jako je *Insert*, *Find* a *RangeQuery*. Třída *cQTree* dědí ze tříd *cPagedTree* a *cQTreeHeader*.

5.2.2.1 metoda Insert

Metoda *Insert* je stejně jako ostatní implementované metody připravena pro práci s n-dimenzionálními souřadnicemi. Jako parametry přebírá ukazatel na uzel reprezentující vkládaný uzel a data, které se do uzlu vloží. Metoda vrací číslo, které reprezentuje stav, s jakým skončilo vkládání uzlu. Číslo je reprezentované statickými konstantami specifikovanými v textu výše. Možné výsledky jsou tyto:

- 0 - uzel úspěšně vložen do stromu bez chyb.
- 1 – totožný uzel jako vkládaný byl již ve stromu nalezen.
- 2 – při vkládání uzlu nastala nespecifikovaná chyba

Pseudokód, vysvětlující funkci vložení uzlu do stromu by mohl vypadat následujícím způsobem:

```
Funkce Insert(vkladanyUzel, data)
{
  priznakVlozeni = UZEL_NEBYL_VLOZEN;
  if(stromJePouzeProCteni) {
    ZobrazUpozorneniUzivateli();
    UkonciAplikaci();
  }
```

```
  indexUzlu = nactiIndexKorenovehoUzlu();
  if(SouradniceVkladanehoUzluNejsouVProstoru()){
    return priznakVlozeni;
  }
  while(priznakVlozeni == UZEL_NEBYL_VLOZEN)
  {
    aktualniUzel = nactiUzel(indexUzlu);
    if(debug_mod = INDIVIDUALNI_VKLADANI){
      VytiskniNaObrazovkuAktualniUzel();
    }
    if(aktualniUzel.VratPocetItemu() == 0){
      aktualniUzel.VlozUzel();
      priznakVlozeni = UZEL_VLOZEN;
    }
    else if(JsouUzlyIdenticke(aktualniUzel,vkladanyUzel)){
      priznakVlozeni == UZEL_JIZ_VLOZEN;
    } else {
      dalsiKvadrant == VypocitejKvadrant(aktualniUzel,vkladanyUzel);
      indexUzlu = aktualniUzel.NactiLink(dalsiKvadrant);
    }
    if(priznakVlozeni == UZEL_NENI_VLOZEN && indexUzlu == PRAZDNY_INDEX)
      novyUzel = VytvorNovyUzel();
      novyUzel.VlozDoUzluData(vkladanyUzel,data);
      aktualniUzel.NastavLink(dalsiKvadrant,novyUzel.NactiUzel);
      UvolniSdilenouCache(novyUzel);
      priznakVlozeni = UZEL_VLOZEN;
    }
    UvolniSdilenouCache(aktualniUzel);
  }
  return priznakVlozeni;
}
```

Následující popis algoritmu se bude vztahovat k pseudokódu, názvy proměnných a metod se v reálném kódu liší.

1. Algoritmus začíná testování, zda je strom určen pouze pro čtení. Tato informace je uložena v proměnné *mReadOnly*, kterou obsahuje předek - třída *cPagedTree*.
2. Načte se uzel kořenového uzlu a uloží do proměnné *indexUzlu*. Kořenový uzel je vytvořen vždy. Vytváří se v metodě *Create*.
3. Otestuje se, zda souřadnice vkládaného uzlu leží v oblasti ohraničené kvadrantovým stromem. Pokud neleží, okamžitě se vrátí obsah proměnné *priznakVlození*, která je v tomto momentu nastavena na hodnotu konstanty *UZEL_NENI_VLOZEN*.
4. Začíná se provádět cyklus while. Jako podmínka cyklu slouží ověření, zda obsahem proměnné *priznakVlození* je hodnota konstanty *UZEL_NENI_VLOZEN*. Je tedy patrné, že se bude provádět, dokud se uzel do stromu nevloží. Pokud podmínka není splněna, pokračuje se bodem 12.

5. Na začátku cyklu while se načte uzel, daný indexem uloženým v proměnné *indexUzlu*.
6. Ověří se, zda je povoleno ladění kódu. Pokud ano, vypíší se souřadnice právě načteného uzlu.
7. Ověří se, zda počet položek (itemů) načteného uzlu je roven nule. Tato podmínka je automaticky splněna při načtení kořenového uzlu, jelikož tento uzel se vytváří jako prázdný, čili bez položek. V tomto případě se vloží vkládaný uzel na toto místo a změní se obsah proměnné *priznakVlozeni* na hodnotu konstanty *UZEL_VLOZEN*.
8. Otestuje se, zda vkládaný uzel je identický s aktuálním uzlem. Pokud ano, do proměnné *priznakVlozeni* se uloží hodnota konstanty *UZEL_JIZ_VLOZEN*.
9. Pokud žádný předchozí test neskončil úspěchem, je třeba se zanořit dále do stromu. Do proměnné *dalsiKvadrant* se uloží číslo kvadrantu, ve kterém leží vkládaný uzel, ve vztahu k aktuálnímu uzlu. Pro dimenzi 2 odpovídá číslo 0 NW kvadrantu, číslo 1 NE kvadrantu, číslo 2 SW kvadrantu a číslo 3 SE kvadrantu. Poté se načte číslo uzlu, na který odkazuje link na pozici dané obsahem proměnné *dalsiKvadrant*. Číslo uzlu se uloží do proměnné *indexUzlu*.
10. V tomto kroku se provede vložení vlastního uzlu. Nejprve se otestuje, zda proměnná *priznakVlozeni* obsahuje hodnotu konstanty *UZEL_NEBYL_VLOZEN*, jelikož to je výchozí hodnota této proměnné a změní se pouze v případě, kdy se uzel vloží do stromu, případně se ve stromě nalezne identický uzel. Současně se testuje, zda načtený *indexUzlu* je prázdný. Pokud budou obě podmínky splněny, nalézáme se v listu stromu a můžeme tedy vložit vkládaný uzel na pozici určenou proměnnou *dalsiUzel*. Nejprve se vytvoří prázdný uzel, na který poté zavoláme metodu *VlozDoUzluData*. Metodě předáme vkládaný uzel, respektive jeho souřadnice a data vkládaného uzlu. Dále je třeba nastavit odkaz na tento vytvořený uzel pomocí metody *NastavLink*, které předáme číslo kvadrantu, ve kterém leží nový uzel a číslo nového uzlu. Poté musíme uvolnit nový uzel ze sdílené cache paměti, kterou používá framework. To provedeme pomocí metody *UvolniSdilenouCache*, které předáme jako parametr pointer ukazující na nový uzel. Jako poslední akce v tomto kroku se provede změna obsahu proměnné *priznakVlozeni* na hodnotu konstanty *UZEL_VLOZEN*.
11. Uvolní se aktuální uzel se sdílené cache paměti opět pomocí metody *UvolniSdilenouCache*, které předáme jako parametr pointer ukazující na aktuální uzel. Zde také končí cyklus while, takže pokud se tedy nenacházíme na listu stromu a nejsou splněny podmínky výše, je třeba se dále zanořit hlouběji do stromu. Pokračuje se bodem 4.
12. Zde se pouze provede vrácení čísla reprezentujícího výsledek metody vložení. Tato hodnota je uložena v proměnné *priznakVlozeni*.

5.2.2.2 ***metoda Find***

Metoda Find je taktéž připravena pro zpracování vícedimenzionálních souřadnic. Vyhledává předaný pointer, ukazující na hledaný uzel a vrátí pravdu, pokud byl nalezen a nepravdu, pokud nebyl nalezen. Algoritmus pracuje podobným způsobem jako u metody *Insert*. Strom se prochází obdobným způsobem a po nalezení uzlu metoda okamžitě vrátí pravdu.

Pseudokód:

```
Funkce Find(hledanyUzel, data)
{
  uzelNalezen = false;
  indexUzlu = nactiIndexKorenovehoUzlu();
  if(SouradniceVkladanehoUzluNejsouVProstoru()){
    return uzelNalezen;
  }
  while(!uzelNalezen)
  {
    aktualniUzel = nactiUzel(indexUzlu);
    if(debug_mod = INDIVIDUALNI_VKLADANI){
      VytiskniNaObrazovkuAktualniUzel();
    }
    if(aktualniUzel.PocetItemu != 0){
      if(JsouUzlyIdenticke(aktualniUzel,vkladanyUzel)){
        uzelNalezen = true;
      } else {
        dalsiKvadrant == VypocitejKvadrant(aktualniUzel,hledanyUzel);
        indexUzlu = aktualniUzel.NactiLink(dalsiKvadrant);
      }
    }
    UvolniSdilenouCache(aktualniUzel);
    if(aktualniUzel.VratPocetItemu() == 0 && indexUzlu == PRAZDNY_INDEX) {
      break;
    }
  }
  return uzelNalezen;
}
```

Popis pseudokódu:

1. Algoritmus načte uzel kořenového uzlu a uloží do proměnné *indexUzlu*. Kořenový uzel je vytvořen vždy. Vytváří se v metodě *Create*.
2. Otestuje se, zda souřadnice vkládaného uzlu leží v oblasti ohraničené kvadrantovým stromem. Pokud neleží, okamžitě se vrátí obsah proměnné *uzelNalezen*, která je v tomto momentu nastavena na *false*.
3. Začíná se provádět cyklus while. Jako podmínka cyklu slouží ověření, zda proměnná *uzelNalezen* je rovna hodnotě *false*. Je tedy patrné, že se bude provádět, dokud se uzel nevyhledá. V tomto případě se cyklus přeruší pouze v bodě 8.

4. Na začátku cyklu while se načte uzel, daný indexem uloženým v proměnné *indexUzlu*.
5. Ověří se, zda je povoleno ladění kódu. Pokud ano, vypíší se souřadnice právě načteného uzlu.
6. Ověří se, zda počet položek (itemů) načteného uzlu není roven nule. U čtení již vložených uzlů by případ, kdy by byl uzen roven nule vůbec neměl nastat. Jediný případ, kdy by tento stav mohl nastat, je při vyhledávání uzlu v prázdném stromu.
    a. Pokud je podmínka splněna, testuje se, zda jsou uzly aktuální uzel a hledaný uzel ekvivalentní. Pokud ano, obsah proměnné *uzelNalezen* se změní na *true*. Zde Je tedy vyhledán vlastní uzel.
    b. Pokud podmínka není splněna, zanořujeme se dále do stromu. Zde je kód totožný jako v metodě *Insert*. Do proměnné *dalsiKvadrant* se uloží číslo kvadrantu, ve kterém leží hledaný uzel, ve vztahu k aktuálnímu uzlu. Čísla uzlů odpovídají kvadrantům jako v metodě *Insert*. Dále se načte číslo uzlu, na který odkazuje link na pozici dané obsahem proměnné *dalsiKvadrant*. Toto číslo se načte do proměnné *indexUzlu*.
7. Uvolní se aktuální uzel ze sdílené cache paměti pomocí metody *UvolniSdilenouCache*, které předáme jako parametr pointer ukazující na aktuální uzel.
8. Testujeme, zda počet položek v aktuálním uzlu je roven nule a taktéž zda *indexUzlu* je prázdný. V tomto bodě se nacházíme v listu stromu a tudíž je jasné, že vyhledání hledaného uzlu proběhlo neúspěšně. Přerušíme tedy cyklus while.
9. Zde se již pouze vrátí hodnota proměnné *uzelNalezen*.

5.2.2.3 ***metoda RangeQuery***

Metoda *RangeQuery* je opět připravena pro práci s vícedimenzionálními souřadnicemi. Jako parametry přebírá dva odkazy na objekt typu *cTuple*, reprezentující levý horní a pravý dolní roh rozsahového dotazu. Metoda vrací ukazatel typu *cItemStream*, který obsahuje veškeré nalezené uzly. Při průchodu stromem není využitá rekurze, ale zásobník, obě metody jsou však ekvivalentní.

Opět následuje pseudokód. Jelikož se v kódu vyskytuje mnoho podmínek if a cyklů while a for, každý delší cyklus, respektive podmínku, jsem pro větší přehlednost označil identifikátorem, který jsem poté uvedl i na konci cyklu resp. podmínky. U krátkých podmínek nebo cyklů jsem identifikátor neuváděl, aby nedošlo k dalšímu zvýšení nepřehlednosti pseudokódu. Jedná se o nejsložitější algoritmus, tudíž jsem se snažil o maximální zjednodušení pseudokódu, kde jsem nechal jen nejdůležitější prvky, ze kterých vyplývá princip tohoto algoritmu.

```
Funkce RangeQuery(parLevyHorniRoh, parPravyDolniRoh)
{
  pocetNalezenychUzlu = 0;
```

```
  spaceLevyHorni, spacePravyDolni, kvadrantLevyHorni, kvadrantPravyDolni =
new cTuple();
  indexUzlu = nactiIndexKorenovehoUzlu();
  zasobnik.InicializujZasobnik();
  if(SouradnicePredanychTupluNejsouVProstoru()){
    return pocetnalezenychUzlu;
  }
  while(!(aktualniHloubka == 0) &&
zasobnik[0][DALSI_KVADRANT_K_PROZKOUMANI] == ZADNY_KVADRANT)
  {//zda zacina while1
    aktualniUzel = nactiUzel(indexUzlu);
    spaceLevyHorni.VlozSouradniceZeZasobniku();
    spacePravyDolni.VlozSouradniceZeZasobniku();
if(zasobnik[aktualniHloubka][DALSI_KVADRANT_K_PROZKOUMANI] !=
ZADNY_KVADRANT)
    {//zde zacina if1
      dalsiKvadrant = zasobnik[aktualniHloubka][
DALSI_KVADRANT_K_PROZKOUMANI];
      while(dalsiKvadrant != ZADNY_KVADRANT)
      {//zde zacina while2
         kvadrantLevyHorni.NastavSouradniceZeZasobniku();
         kvadrantPravyDolni.NastavSouradniceZeZasobniku();
         if(ObdelnikySeProtinaji(kvadrantLevyHorni, kvadrantPravyDolni,
parLevyHorniRoh, parPravyDolniRoh))
         {//zde zacina if2
           indexUzlu = aktualniUzel.NactiLink(dalsiKvadrant);
            if(indexUzlu != PRAZDNY_INDEX)
            {
              VlozUdajeDoZasobniku();
              aktualniHloubka ++;
              VlozUdajeDoZasobniku();
              UvolniSdilenouCache(aktualniUzel);
              break;
            } else {
              dalsiKvadrant ++;
              VlozUdajeDoZasobniku();
            }
         } else {
            dalsiKvadrant ++;
            VlozUdajeDoZasobniku();
        }//zde konci if2
      }//zde konci while2
      if(dalsiKvadrant == ZADNY_KVADRANT)
      {//zde zacina if3
        if(UzelLeziVOBdelniku(parLevyHorniRoh, parPravyDolniRoh,
aktualniUzel))
         {
           vysledek.Pridej(aktualniUzel);
         }
```

```
          if(aktualniHloubka != 0)
          {
            aktualniHloubka --;
          }
        }//zde konci if3
      } else { //zde pokracuje if1
        if(UzelLeziVOBdelniku(parLevyHorniRoh, parPravyDolniRoh,
aktualniUzel))
        {
          vysledek.Pridej(aktualniUzel);
        }
        if(aktualniHloubka != 0)
        {
          aktualniHloubka --;
        }
      }//zde konci if1
    }//zde konci while 1
    UvolniPamet();
    return pocetNalezenychUzlu;
```

Popis pseudokódu:

1. Nejprve dojde k deklaraci a inicializaci dočasných proměnných, do kterých se ukládá počet nalezených uzlů, číslo dalšího kvadrantu k prozkoumání, aktuální hloubka apod.
2. Dojde k vytvoření dočasných n-tic (typu *cTuple*), které se ovšem nealokují dynamicky, ale vytvoří se oblast sdílené paměti (memory pool), do které se vloží dočasné n-tice.
3. Proběhne inicializace zásobníku, reprezentovaného dvourozměrným polem. Do tohoto zásobníku se ukládá číslo uzlu, číslo dalšího kvadrantu, který je nutné prozkoumat a n-tice ohraničující prostor aktuálního kvadrantu. Pro první úroveň je zřejmé, že n-tice budou ohraničovat prostor celého stromu. Je tedy nutné nastavit toto ohraničení. Všechny další hodnoty budou nastaveny na 0.
4. Zde začíná cyklus while, v pseudokódu označen jako while1. Je to hlavní cyklus celého algoritmu. Cyklus skončí v momentě, kdy se program bude nacházet na nulté úrovni a zároveň prozkoumal v této nulté úrovni všechny kvadranty.
    a. Proběhne načtení uzlu, který je dán obsahem proměnné *indexUzlu*.
    b. Do n-tic, které reprezentují okraje aktuálního prostoru se vloží hodnoty ze zásobníku.
    c. V podmínce if (v pseudokódu if1) se kontroluje, zda pro aktuální hloubku existuje další kvadrant k prozkoumání.
        i. Pokud ano, načte se ze zásobníku číslo dalšího kvadrantu, které je nutné prozkoumat.

ii. Následuje cyklus while (v pseudokódu while2). Tento cyklus se bude provádět, dokud budou existovat další kvadranty k prozkoumání (v aktuální hloubce).
    1. Pro n-tice, reprezentující aktuální kvadrant, se vypočtou okraje.
    2. V podmínce if (v pseudokódu if2) se kontroluje, zda je v průniku aktuální zkoumaný kvadrant a okno rozsahového dotazu.
        a. Pokud ano, do proměnné *indexUzlu* je načten odkaz na uzel, reprezentující další kvadrant.
            i. V tomto bodu dochází k zanořování hlouběji do stromu. Pokud *indexUzlu* není prázdný, znamená to, že program není ještě na konci stromu a je třeba se zanořit hlouběji. Do zásobníku se na pozici, dané aktuální hloubkou, uloží číslo právě zpracovávaného uzlu a inkrementované číslo kvadrantu k prozkoumání. Dále se inkrementuje aktuální hloubka. Číslo kvadrantu v následující úrovni se nastaví na nulu. Do zásobníku se na pozice, reprezentující hranice prostoru zapíší souřadnice z n-tic, reprezentujících aktuální kvadrant. Dále se už jen uvolní zámek na aktuální uzel.
            ii. Pokud *indexUzlu* je prázdný, inkrementuje se číslo dalšího kvadrantu k prozkoumání a toto číslo se zapíše i do zásobníku.
        b. Pokud ne, provede se stejný krok jako v bodu ii o odstavec výše.

iii. V podmínce if (v pseudokódu if3) se ptáme, zda je program ve stavu, kdy už prozkoumal všechny možné kvadranty. Tento stav nastává tehdy, pokud se program nachází v listovém uzlu a nemá se tedy kam zanořovat. V tomto momentu proběhne metoda, která ověří, zda se aktuální uzel nachází v okně rozsahového dotazu. Pokud ano, vloží se aktuální uzel do výsledku reprezentovaného ukazatelem typu *cItemStream*. Dále dekrementuje aktuální hloubku, jelikož teď se bude program vynořovat nahoru. Uvolní se zámek na aktuální uzel a do

proměnné *indexUzlu* se načte předek uzlu, který je uložený ve speciálním zásobníku.

iv. Pokud podmínka if1 nebyla splněna, provedou se stejné operace jako v bodu iii v odstavci výše.

5. Proběhne vyprázdnění paměti a metoda vrátí počet nalezených uzlů.

5.2.2.4 ***metoda IsInSpace***

Tato metoda slouží pro ověření, zda souřadnice předaného uzlu leží v oblasti určené kvadrantovým stromem. V odborné publikaci, ze které jsem čerpal [1], je tento pojem nazván *underlying space*. Jako parametry přebírá pouze odkaz na danou položku, jejíž souřadnice chceme otestovat. Metoda vrací *true* pokud souřadnice uzlu leží v dané oblasti, *false* pokud tam neleží.

Pseudokód:

```
Funkce IsInSpace(testovanaPolozka)
{
  dimenze = nactiDimenzi();
  for(int i; i < dimenze; i++){
    if(!(0 < testovanaPolozka.NactiSouradnici(i) < UINT_MAX)) {
      return false;
    }
  }
  return true;
}
```

Popis pseudokódu:

1. Nejprve proběhne načtení dimenze, se kterou se bude pracovat. V reálném kódu se dimenze získává z ukazatele typu *cSpaceDescriptor*, jedná se tedy o popisovač daného prostoru. Dimenze se uloží do proměnné *dimenze*.
2. Zde začíná cyklus for. Provede se tolikrát, jak velká je dimenze prostoru. Definuje se zde proměnná *i* a inkrementuje se po každém projití cyklu.
   a. V cyklu for se ověřuje, zda souřadnice na *i*-té pozici nejsou záporné a zároveň jsou menší než maximální hodnota unsigned int, což je datový typ souřadnic uzlu. Souřadnice se ve skutečnosti získává přes statickou metodu *GetUint* třídy *cTuple*. Této metodě je třeba předat výsledek metody *GetData* aplikované na testovanou položku, pořadí souřadnice a pointer ukazující na aktuální space descriptor. Pokud podmínka není splněna, okamžitě se vrátí hodnota *false*.
3. Vrátí se návratová hodnota *true*.

5.2.2.5 ***metoda IsInNode***

Metoda *IsInNode* testuje, zda jsou dané body ekvivalentní. V praxi funguje tak, že porovnává vždy dvě souřadnice předaných uzlů na stejné pozici. Pokud se alespoň jedna souřadnice liší, vrátí

false. Pokud tedy budou uzly stejné, respektive budou mít shodné souřadnice, metoda vrátí *true*. Jako parametry se metodě předávají pointer ukazující na aktuální uzel a odkaz na vkládaný uzel.

Pseudokód:

```
Funkce IsInNode(aktualniUzel, vkladanyItem)
{
  dimenze = spaceDescriptor.VratDimenzi();
  for(int i = 0; i < dimension; i++)
  {
    if(aktualniUzel.NactiSouradnici(i) != vkladanyItem.NactiSouradnici(i))
    {
              return false;
    }
  }
}
```

Popis pseudokódu:

1. Nejprve opět proběhne načtení dimenze prostoru, která se v reálném kódu získá z ukazatele typu *cSpaceDescriptor*. Dimenze prostoru se uloží do proměnné *dimenze*.
2. Poté se začne provádět cyklus for, který je identický jako u metody *IsInSpace*. Vytvoří se proměnná *i* a inkrementuje se po každém projití cyklu až do velikosti dimenze.
   a. V cyklu for je jediný krok a to kontrola, zda jsou souřadnice aktuálního a vkládaného uzlu stejné na *i*-té pozici. Pro tuto operaci byla využita statická metoda *Equal* třídy *cTuple*. Této metodě je třeba předat návratovou hodnotu metody *GetItem(0)*, která je aplikována na *aktualniUzel*. Číslo 0 v tomto případě značí, že se pracuje s itemem na pozici 0, protože v této implementaci kvadrantových stromů obsahuje uzel vždy jen jeden item – instanci třídy *cTuple*. Dále se metodě *Equal* předá výsledek metody *GetData()*, kterou aplikujeme na *vkladanyItem*, pořadí souřadnice a pointer ukazující na aktuální space descriptor. Pokud podmínka není splněna, vrací metoda okamžitě hodnotu *false*.
3. Vrátí se návratová hodnota *true*.

*5.2.2.6 **metoda ComputeQuadrant***

V této metodě se provádí výpočet kvadrantu, ve kterém leží předaný uzel ve vztahu k aktuálnímu uzlu. Jako parametry slouží pointer ukazující na aktuální uzel a odkaz na vkládaný item. Metoda vrací číslo, reprezentující kvadrant, ve kterém se vkládaný item nachází.

Pseudokód:

```
Funkce ComputeQuadrant(aktualniUzel, vkladanyItem)
{
  dimenze = spaceDescriptor.VratDimenzi();
  if(debug_mod = INDIVIDUALNI_VKLADANI){
    VytiskniNaObrazovkuAktualniUzel();
  }
```

```
  for(int i = 0; i < dimension; i++)
  {
    if((aktualniUzel.NactiSouradnici(i) < vkladanyItem.NactiSouradnici(i))
    {
      bit = false;
    }
    vysledek = NastavBit(i, bit)
  }
  return vysledek;
}
```

Popis pseudokódu:

1. Nejprve opět proběhne načtení dimenze prostoru, která se v reálném kódu získá z pointeru typu *cSpaceDescriptor*. Dimenze prostoru se uloží do proměnné dimenze.
2. Pokud je aktivován debug mód pro individuální vkládání, vypíší se souřadnice aktuálního uzlu na obrazovku.
3. Poté se začne provádět cyklus for, který je identický jako u metod *IsInSpace* a *IsInNode*. Vytvoří se proměnná *i* a inkrementuje se po každém projití cyklu až do velikosti dimenze.
    a. V cyklu for se nejprve zkontroluje, zda *i*-tá souřadnice aktuálního uzlu je menší než *i*-tá souřadnice vkládaného itemu. Pokud ano, nastaví se dočasná lokální proměnná bit typu bool na hodnotu *false*. Toto je provedeno pomocí statické metody *Equal* třídy *cTuple*. Pokud tato metoda vrací hodnotu -1, pak je *i*-tá souřadnice aktuálního uzlu menší než souřadnice vkládaného itemu na stejné pozici. Kontroluje se tedy, zda výsledek této metody je menší než 0.
    b. Na konci cyklu for se proměnná *vysledek* nastaví na hodnotu, která je uložená v proměnné bit. Použita je statická metoda *SetBit* třídy *cBitStringNew*. Této metodě je třeba předat pointer typu char ukazující na odkaz na proměnnou *vysledek*. S touto proměnnou se tedy ve výsledku pracuje, její hodnota je měněna. Dále je třeba předat pozici souřadnice a hodnotu, na kterou má metoda proměnnou nastavit. Smysl je v tom, že proměnná *vysledek*, i když je typu unsigned integer, je chápána jako datový typ bitstring, respektive bitové pole. Na každé pozici tohoto pole je bit, který se nastavuje po každém projití cyklu for. Nakonec se bitové pole, ve kterém je uložen výsledek v binární soustavě, překonvertuje na celé číslo. Toto řešení je použito především pro to, aby metoda vracela korektní výsledky i pro n-tice v dimenzi větší než 2. Proto nelze použít standardní variantu s počítáním a porovnáváním souřadnic popsanou v kapitole 3. Jednoduché porovnání souřadnic a podle výsledku porovnávání vrácení hodnot by fungovalo pouze pro dimenzi 2.
4. Metoda vrátí proměnnou *vysledek*.

5.2.3 Třída cQTreeHeader

Tato třída reprezentuje hlavičku stromu. Obsahuje metody nutné pro inicializaci stromu, práci se space descriptorem (popisovačem prostoru), čtení a zápis do paměti, nastavení základních údajů o datech se kterými se bude pracovat a podobně. Třída *cQTreeHeader* dědí z třídy *cTreeHeader*.

5.2.3.1 ***metoda Init***

Metoda *Init* slouží pro prvotní nastavení stromu, vnitřní struktury uzlů, názvu stromu apod. Metoda *Init* je z hlavní metody volána s parametry space descriptor, velikost klíče uzlu a velikost dat uzlu.

1. Nejprve se provede volání metody *Init* bez parametrů, kde se nastaví název stromu, verze stromu a buildu a inicializují se časovače.
2. Dále se zavolá metoda *SetNodeHeaderCount* třídy *cDStructHeader* kde se do proměnné *mNodeHeaderCount* nastaví počet hlaviček stromu. V našem případě se pracuje s jednou hlavičkou. Dále se v metodě *SetNodeHeaderCount* vytváří pole hlaviček a pole identifikátorů těchto hlaviček.
3. Vytvoří se instance hlavičky s parametry velikosti klíče a velikosti dat.
4. Hlavičce se nastaví space descriptor, čili specifikuje se, v jakém prostoru se bude daný vícedimenzionální strom nacházet.
5. K danému stromu se přiřadí hlavička pomocí metody *SetNodeHeader* třídy *cDStructHeader*

5.2.3.2 ***metoda HeaderSetup***

V této metodě se provádí základní nastavení hlavičky stromu. Specifikují se velikosti uzlů, počet různých druhů položek v uzlech a podobně.

1. Nejprve se nastavuje počet extra itemů v uzlu, mimo samotné souřadnice. V případě kvadrantových stromů není třeba dalších itemů, čili je tento počet nastaven na 0. Nastavení je provedeno pomocí metody *SetLeafNodeExtraItemCount*, nacházející se ve třídě *cTreeHeader*.
2. Dále se nastavuje počet itemů v uzlu. Jak již bylo řečeno, item, neboli položku, reprezentuje instance třídy *cTuple*, ve které jsou uloženy souřadnice uzlu. V implementaci se z pohledu frameworku pracuje s každým uzlem jako s listovým uzlem, čili je toto nastavení provedeno pomocí metody *SetLeafNodeItemCapacity* třídy *cTreeHeader*.
3. Poté se nastavuje počet odkazů na další uzly. Je zřejmé, že pokud má strom pracovat s dimenzí větší než 2, není možné vytvořit pevný počet odkazů. Nejprve se tedy vypočte počet linků pro danou dimenzi, která je dána vztahem 2^d, kde d značí dimenzi prostoru.

Výsledek se poté předá metodě *SetLeafNodeFanoutCapacity* třídy *cTreeHeader*, kde se již nastaví korektní počet odkazů.

4. Na vytvořenou instanci typu *cNodeHeader*, přetypovanou na *cQTreeNodeHeader* se zavolá metoda *ComputeNodeCapacity*, který je popsána níže.

5.2.4 Třída cQTreeNode

Tato třída reprezentuje jak vnitřní, tak listový uzel stromu a obsahuje pouze prázdný konstruktor a destruktor. Veškeré potřebné metody tedy dědí ze třídy *cFCNode*, kterou obsahuje framework DBRG.

5.2.5 Třída cQTreeNodeHeader

Třída cQtreeNodeHeader reprezentuje hlavičku uzlu. Obsahuje potřebné metody pro práci s uzlu, jako *WriteNode*, *ReadNode*, sloužící pro zápis, respektive čtení uzlu z paměti. Dále obsahuje metodu *CopyNode* a *CreateCopy* pro kopírování uzlu a taktéž důležitou metodu *ComputeNodeCapacity*.

5.2.5.1 metoda ComputeNodeCapacity

Tato metoda slouží pro výpočet reálné velikosti uzlu v paměti.

1. Nejprve se provede načtení velikosti indexu, čili odkazu na další uzel.
2. Provede se výpočet velikosti uzlu, založeném na velikosti položky (proměnná *itemSize*), dále velikosti odkazu (*linkSize*), jejich počtu v uzlu (v našem případě obsahuje struktura vždy pouze 1 item, počet odkazů se mění v závislosti na dimenzi).
3. Zavolají se metody pro nastavení potřebných parametrů (*SetNodeCapacity*, *SetNodeSerialSize*).
4. Nakonec se zavolá metoda *SetInMemOrders* pro vytvoření hranic mezi položkami uzlu.

5.2.6 Třída cQTree.cpp

Tato třída slouží pro vlastní běh aplikace, obsahuje metodu *main*, vytvoření instance stromu apod. Obsahuje některé důležité globální proměnné například:

- dim – dimenze prostoru stromu, pro změnu dimenze tedy stačí změnit tuto proměnnou a celý strom pak bude pracovat v nastavené dimenzi.
- count – počet uzlů ve stromu.
- cacheSizeInNode – počet uzlů, uchovávaných v operační paměti. Pokud tedy počet uzlů stromu překročí tento počet, začnou se ukládat uzly na pevný disk.

Dále třída obsahuje několik základních metod:

- Init() – incializace stromu.
- CreateDataCollection() – vygenerování souřadnic jednotlivých uzlů.

- InsertData() – vložení uzlů s vygenerovanými souřadnicemi do stromu.
- PointQueryData() – vyhledávání vygenerovaných uzlů ve stromu.
- RangeQueryData() – provedení rozsahového dotazu
- Clear() – odstranění instance stromu, uzlů apod. z paměti. Korektní ukončení dočasné databáze.

5.2.6.1 ***metoda Init***

Metoda *Init* slouží pro vytvoření a inicializaci stromu.

1. Nejprve se vytvoří dočasná databáze na disku, reprezentována instancí třídy *cQuickDB*.
2. Na hlavičku kvadrantového stromu (instance třídy *cQTreeHeader* s parametrem *cTuple*) se zavolá metoda *HeaderSetup*.
3. Vytvoří se samotný strom – na instanci třídy *cQTree* s parametrem *cTuple* se zavolá metoda *Create.*

5.2.6.2 ***metoda CreateDataCollection***

Tato metoda slouží pro vygenerování potřebných n-tic, jejichž počet závisí na nastavení proměnné *count*. Využívá se třídy *cGaussRandomGenerator*. Vygenerované n-tice se ukládají do pole typu *cTuple*.

5.2.6.3 ***metoda InsertData***

Zde se prování vkládání samotných uzlů do stromu. Prochází se pole n-ntic vygenerovaných v metodě *CreateDataCollection* a postupně se vkládají do stromu.

1. Aktivuje se časovač pro měření času vkládání.
2. Ukládání probíhá v cyklu for, který je omezen počtem n-tic. Na začátku cyklu proběhne výpis počtu již vložených uzlů (po určitém intervalu, například po každých 10 000 vložených uzlů)
3. Na objekt *tree*, reprezentující daný strom (instance třídy *cQTree*) se zavolá metoda *Insert*, návratová hodnota se uloží do nově vytvořené proměnné *insertFlag* typu integer.
4. Dále se inkrementují počítadla. Pokud proběhlo vložení úspěšně, inkrementuje se počet vložených uzlů. Pokud neproběhlo úspěšně, přičemž důvodem bylo, že se daný uzel ve stromu již nacházel, inkrementuje se počítadlo duplikátních uzlů. V každém jiném případě se inkrementuje počítadlo chyb, přičemž se vypíše chybová hláška.
5. Zastaví se časovač. Vypíše se celkový čas vkládání a počet vložených uzlů za sekundu.

5.2.6.4 ***metoda PointQueryData***

Tato metoda je určena pro hledání vložených uzlů. Opět se prochází pole vygenerovaných n-tic a u všech probíhá vyhledání ve stromu. Průběh se téměř totožný s metodou *InsertData*. Pokud

nastane případ, kdy se n-tice nachází ve vygenerovaném poli n-tic, ale není nalezena ve stromu, vypíše se chybová hláška a program se ukončí. Jedná se totiž o fatální chybu, která by neměla v běžném provozu nastat. Opět se zde používá časovač pro měření času vyhledávání.

5.2.6.5 ***metoda RangeQueryData***

V této metodě se vytvoří n-ntice, reprezentující levý horní a pravý dolní okraj okna rozsahového dotazu. Na objekt *tree* se zavolá metoda *RangeQuery*, přičemž té se předají vytvořené n-tice. Opět se zde používají časovače a čítače vyhledaných uzlů v rozmezí rozsahového okna.

5.2.6.6 ***metoda Clear***

Tato metoda slouží pro odstranění využitých objektů z paměti a zavření a smazání dočasné databáze.

5.2.6.7 ***metoda Main***

Je patrné, že tato metoda se spouští jako první. Obsahuje posloupnost volání jednotlivých procedur. Metody se volají v tomto pořadí:

1. Init()
2. CreateDataCollection()
3. InsertData()
4. PointQueryData()
5. RangeQueryData()
6. Clear()

5.3 Problémy při implementaci

První problém nastal, když jsem přizpůsoboval metodu *ComputeQuadrant* pro obecnou dimenzi. Využil jsem instanci třídy *BitString*, kde je možné uložit číslo v binární formě. Nicméně pak se zpomalilo vkládání a vyhledávání uzlu (obě tyto metody využívají metodu *ComputeQuadrant*) zhruba desetinásobně. Problém jsem vyřešil tak, že jsem nevytvářel instance třídy *BitString*, ale využíval jsem pouze statické metody této třídy.

Další problém nastal v metodě *RangeQuery*, která byla příliš náročná na paměť. V metodě se původně vytvářely instance třídy *cTuple*, reprezentující dočasné n-tice. Nicméně paměť, kterou zabíraly n-tice se po ukončení metody nevracela. Situaci jsem řešil pomocí instance metody *cMemoryPool*, kterou nabízí framework DBRG. Tato instance reprezentuje oblast sdílené paměti, do které se poté uloží ukazatele na jednotlivé n-tice, přičemž při každém vložení se vyhradí právě velikost n-tice. S n-ticí se pak pracuje naprosto stejně jako s instancí třídy *cTuple*. Na konci metody

RangeQuery se na instanci *cMemoryPool* zavolá metoda *FreeMem*, která uvolní paměť, kterou zabíraly n-tice.

6 Testování výkonnosti

Tato část kapitoly je věnována testování výkonnosti kvadrantových stromů a obsahuje též srovnání výsledků s datovou strukturou R-strom. Otestovány jsou základní operace, jako vytváření indexu (čili vkládání uzlů do stromu), bodové dotazy a rozsahové dotazy. Testy jsou provedeny se dvěma druhy dat. První čtyři testy v každé kategorii jsou realizovány s kolekcí s pseudonáhodně vygenerovaných dat. Kolekce dat pro první test obsahuje 1 000 000 uzlů, pro druhý test pak 8 000 000 uzlů. Testy s náhodnými daty jsou provedeny pro dimenzi 2 a 5. Poslední test je proveden pro kolekci reálných dat, která obsahuje 1 000 000 uzlů a je dimenze 3. Rozsahové dotazy jsou testovány pro dimenzi 3. U veškerých testů se s uzly pracovalo v operační paměti počítače.

Každý test jsem provedl třikrát a výsledky zprůměroval. Veškeré časy jsou tedy uvedeny jako již zprůměrované.

Na začátku prvních dvou testů se provede vygenerování pseudonáhodných dat. Toto generování se vykonává v metodě *CreateDataCollection* třídy *QTree*. Čísla jsou generována podle Gaussova rozdělení pravděpodobnosti.

6.1.1 Vytváření indexu

Zde se provádí testování výkonnosti vkládání uzlů do stromu. Vkládání je realizováno v metodě *InsertData* třídy *CQTree*. V metodě se prochází pole vygenerovaných n-tic a postupně se vkládají do stromu. Konkrétně metoda funguje následovně:

1. Aktivuje se časovač pro měření času vkládání.
2. Ukládání probíhá v cyklu for, který je omezen počtem n-tic.
    a. Na objekt *tree*, reprezentující daný strom (instance třídy *cQTree*) se zavolá metoda *Insert*, návratová hodnota se uloží do nově vytvořené proměnné *insertFlag* typu integer.
    b. Volitelně je možné zkontrolovat, zda daná n-tice byla skutečně do stromu vložena a tudíž zavolat ihned na strom metodu *Find*. Nicméně toto je ve výchozím nastavení deaktivováno.
3. Zastaví se časovač. Jsou vypsány statistiky, které obsahují čas vkládání a počet vložených uzlů za sekundu.

6.1.1.1 Kolekce s náhodnými daty

Nejprve se tedy testovaly kolekce s náhodnými daty a s velikostí kolekce 1 000 000.

Test vkládání, počet uzlů = 1 000 000, dimenze = 2			
Typ stromu	*délka vkládání [s]*	*počet insertů za sekundu*	*výška stromu*
Kvadrantový strom	*6,47*	*154464*	*27*
R-strom	*8,82*	*113327*	*3*

Test vkládání, počet uzlů = 1 000 000, dimenze = 5			
Typ stromu	*délka vkládání [s]*	*počet insertů za sekundu*	*výška stromu*
Kvadrantový strom	*3,04*	*329056*	*11*
R-strom	*15,385*	*64998*	*3*

Poté se testovala kolekce obsahující 8 000 000 uzlů.

Test vkládání, počet uzlů = 8 000 000, dimenze = 2			
Typ stromu	*délka vkládání [s]*	*počet insertů za sekundu*	*výška stromu*
Kvadrantový strom	*76,49*	*104591*	*32*
R-strom	*84,26*	*94945*	*3*

Test vkládání, počet uzlů = 8 000 000, dimenze = 5			
Typ stromu	*délka vkládání [s]*	*počet insertů za sekundu*	*výška stromu*
Kvadrantový strom	*34,36*	*232822*	*13*
R-strom	*246,76*	*32421*	*4*

6.1.1.2 ***Kolekce s reálnými daty***

Nakonec se testovala kolekce s reálnými daty, která obsahovala 1 000 000 uzlů. Tato kolekce byla předpřipravena v textovém souboru. Pro čtení z tohoto souboru jsem využil třídu *cTuplesGenerator*, která obsahuje metody pro čtení ze souboru, vrácení další n-tice apod.

Test vkládání, počet uzlů = 1 000 000, dimenze = 3			
Typ stromu	*délka vkládání [s]*	*počet insertů za sekundu*	*výška stromu*
Kvadrantový strom	*121,22*	*8249*	*2594*
R-strom	*36,73*	*27223*	*3*

6.1.2 **Bodové dotazy**

Další částí testu bylo zjišťování výkonnosti struktury při bodových dotazech. Bodové dotazy jsou implementovány v metodě *PointQueryData* ve třídě *CQTree*. Tato část má velmi podobný průběh jako u vkládání, místo metody *Insert* se volá metoda *Find*, přičemž n-tice se nahrávají z vygenerovaného pole n-tic. Postupně se prochází celé pole n-tic o počtu 1 000 000, respektive 8 000 000 a tyto n-tice se vyhledávají ve stromu.

6.1.2.1 ***Kolekce s náhodnými daty***

Opět je nejdříve testována kolekce s náhodnými daty a s jedním milionem uzlů.

Test bodových dotazů, počet uzlů = 1 000 000, počet bodových dotazů = 1 000 000, dimenze = 2			
Typ stromu	*délka dotazování [s]*	*počet bodových dotazů za sekundu*	*výška stromu*
Kvadrantový strom	*6,51*	*153515*	*27*
R-strom	*6,16*	*162285*	*3*

Test bodových dotazů, počet uzlů = 1 000 000, počet bodových dotazů = 1 000 000, dimenze = 5			
Typ stromu	*délka dotazování [s]*	*počet bodových dotazů za sekundu*	*výška stromu*
Kvadrantový strom	*2,69*	*372440*	*11*
R-strom	*28,94*	*34557*	*3*

Poté je otestována kolekce o velikosti 8 000 000.

Test bodových dotazů, počet uzlů = 8 000 000, počet bodových dotazů = 8 000 000, dimenze = 2			
Typ stromu	*délka dotazování [s]*	*počet bodových dotazů za sekundu*	*výška stromu*
Kvadrantový strom	*75,51*	*105941*	*32*
R-strom	*78,46*	*101964*	*3*

Test bodových dotazů, počet uzlů = 8 000 000, počet bodových dotazů = 8 000 000, dimenze = 5			
Typ stromu	*délka dotazování [s]*	*počet bodových dotazů za sekundu*	*výška stromu*
Kvadrantový strom	*30,56*	*261754*	*13*
R-strom	*599,93*	*13335*	*4*

6.1.2.2 ***Kolekce s reálnými daty***

Kolekce s reálnými daty byla testována taktéž.

Test bodových dotazů, počet uzlů = 1 000 000, počet bodových dotazů = 1 000 000, dimenze = 3			
Typ stromu	*délka dotazování [s]*	*počet bodových dotazů za sekundu*	*výška stromu*
Kvadrantový strom	*120,06*	*8329*	*2594*
R-strom	*5,09*	*196657*	*4*

6.1.3 **Rozsahové dotazy**

Posledním testem bylo měření výkonnosti rozsahových dotazů. Procedura je implementována v metodě *RangeQueryData* třídy *CQTree*. U dotazů se navíc uchovává celkový počet vrácených uzlů. U každého testu se rozsahový dotaz provede 1000x.

1. Nejprve se aktivuje časovač pro měření času.
2. Vytvoří se dvě instance třídy *cTuple*, reprezentující levý horní a pravý dolní roh rozsahového okna.
3. Poté se prochází cyklus for.

a. Do n-tic se vloží hodnoty souřadnic, které jsou dány počáteční hodnotou, dimenzí a počtem již prošlých cyklů. Velikost souřadnic tedy roste lineárně, negenerují se náhodně.
b. Na instanci stromu nazvanou *tree* se zavolá metoda *RangeQuery*, kde se jako parametry předávají nově vytvořené n-tice. Z návratové hodnoty metody *RangeQuery* se získá počet vrácených uzlů, který se přičítá do proměnné *rangeQueryResultsCount*, kde se uchovává celkový počet nalezených uzlů.

4. Zastaví se časovač. Jsou vypsány statistiky, které obsahují čas provádění rozsahového dotazu, počet dotazů za sekundu a také celkový počet vrácených uzlů.

6.1.3.1 ***Kolekce s náhodnými daty***

Nejprve se rozsahové dotazy testují na náhodných datech. Velikost kolekce je opět 1 000 000.

Test rozsahových dotazů, počet uzlů = 1 000 000, počet rozsahových dotazů = 1000, dimenze = 3			
Typ stromu	*délka dotazování [s]*	*počet rozsahových dotazů za sekundu*	*Průměrná velikost výsledku na 1 dotaz*
Kvadrantový strom	*0,22*	*4505*	*14,772*
R-strom	*0,17*	*6024*	*14,772*

Poté se rozsahové dotazy otestovaly na kolekci velikosti 8 000 000.

Test rozsahových dotazů, počet uzlů = 8 000 000, počet rozsahových dotazů = 1000, dimenze = 3			
Typ stromu	*délka dotazování [s]*	*počet rozsahových dotazů za sekundu*	*Průměrná velikost výsledku na 1 dotaz*
Kvadrantový strom	*0,54*	*1848*	*99,184*
R-strom	*0,25*	*4000*	*99,184*

6.1.3.2 ***Kolekce s reálnými daty***

I zde bylo provedeno testování na kolekci reálných dat.

Test rozsahových dotazů, počet uzlů = 1 000 000, počet rozsahových dotazů = 1000, dimenze = 3			
Typ stromu	*délka dotazování [s]*	*počet rozsahových dotazů za sekundu*	*Průměrná velikost výsledku na 1 dotaz*
Kvadrantový strom	*24,50*	*40,8*	*122,782*
R-strom	*0,17*	*5882*	*122,782*

6.1.4 **Zhodnocení výsledků**

Kvadrantový strom převyšuje R-strom v rychlosti vkládání a bodových dotazech. Výkon je u dimenze 5 vyšší téměř pětkrát při vkládání. Rozdíl výkonu navíc s rostoucím počtem uzlů nadále roste. Výška kvadrantového stromu byla v těchto případech 11 pro 1 000 000 bodů a 13 pro 8 000 000 bodů. Výška R-stromu byla 3 pro 1 000 000 bodů a 4 pro 8 000 000 bodů.

Ovšem velmi záleží na dimenzi bodů, jelikož u malé dimenze, v tomto testování 2, stoupala výška kvadrantového stromu. U 1 000 000 bodů vzrostla výška na 27, kdežto u R-stromu se drží výška

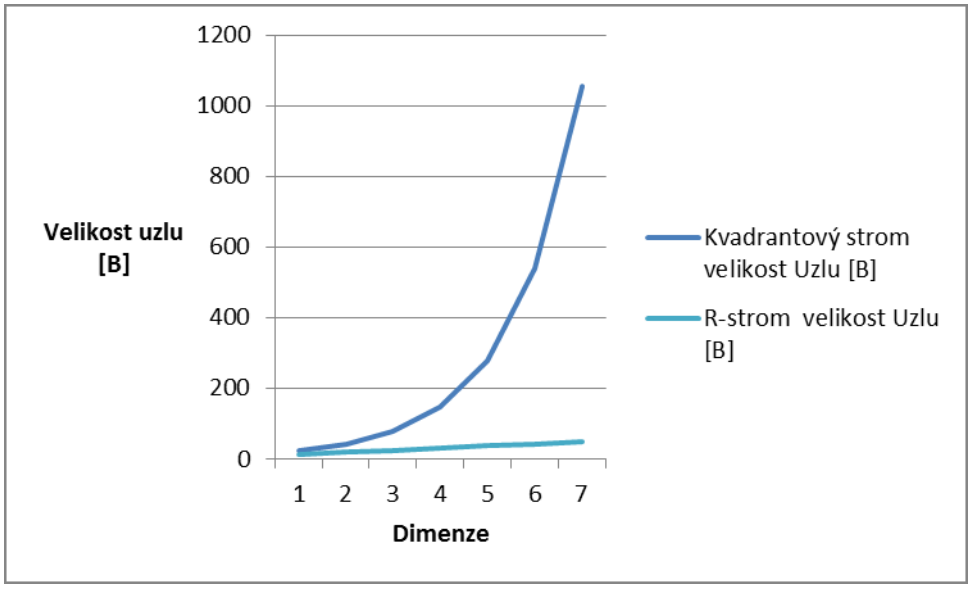
na úrovni 3 jak pro 1 000 000, tak pro 8 000 000 bodů. V těchto případech je vkládání a bodové dotazy u kvadrantového stromu stále rychlejší, nicméně rozdíl už činí pouze několik procent.

Nevýhoda kvadrantových stromů spočívá také ve velké prostorové složitosti takto vytvořeného stromu. Velikost uzlu kvadrantového stromu je dána vztahem $velikostUzlu = velikostN - tice + \left(2^d * velikostOdkazuNaDalsiUzel\right)$, kde d značí dimenzi [4]. Velikost n-tice je v reálném testu 20 B, velikost odkazu na další uzel je rovna 4 B, čili velikost uzlu je při dimenzi 5 rovna 148 B. Při jednom milionu vložených bodů se dostáváme na hodnotu 148 MB. Na obrázku 5.1 je zobrazena závislost velikosti uzlu na dimenzi prostoru.

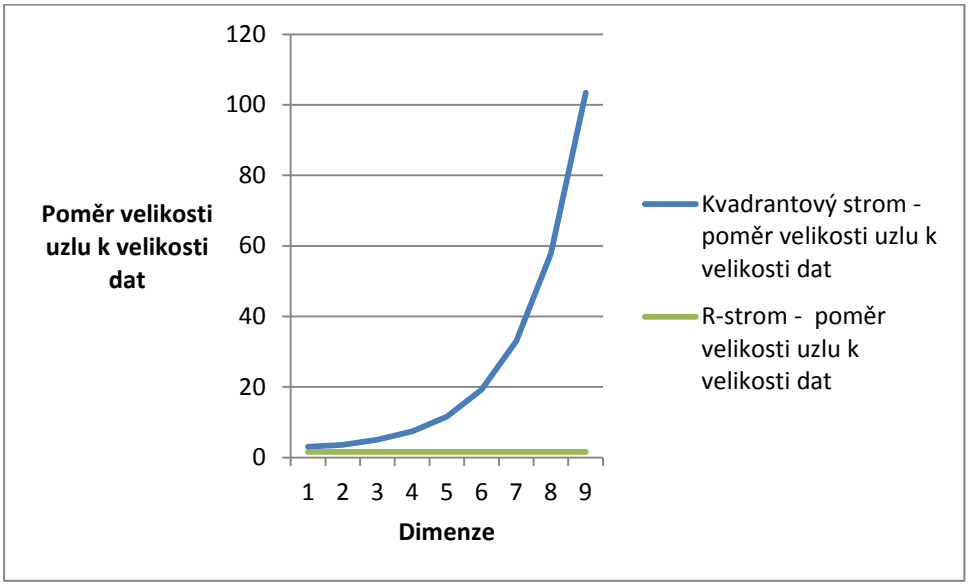
Důvod strmého nárůstu velikosti uzlu kvadrantového stromu spočívá v tom, že uzel musí mít alokované místo v paměti pro všechny odkazy na své děti. Při dimenzi 2 jsou tyto odkazy pouze 4, nicméně už při dimenzi 5 je těchto odkazů 32. Což při velikosti odkazu 4 B dává 128 B jen pro odkazy, přičemž samotná n-tice zabírá pouze 20 B.

Oproti kvadrantovému stromu je prostorová složitost R-stromu mnohem nižší a hlavně nemá tak strmý nárůst velikosti uzlu při rostoucí dimenzi. Při výpočtu velikosti uzlu využíváme průměrné hodnoty, jelikož neznáme přesný počet vnitřních uzlů. Velikost uzlu je tedy dána vztahem $velikostUzlu = velikostN - tice * \left(\frac{1}{0,7}\right) * (1,1)$ [4]. Poměr $\left(\frac{1}{0,7}\right)$ znamená utilizaci 70% a odhadujeme, že průměrný počet vnitřních uzlů se bude pohybovat kolem 10%, čili výsledek vynásobíme číslem 1,1. Velikost uzlu se tedy s rostoucí dimenzí zvyšuje velmi pozvolna.

Na obrázku 5.2 vidíme velikost poměru mezi celkovou velikostí uzlu a velikostí dat (čili n-tice souřadnic). U R-stromu tento poměr nezávisí na dimenzi prostoru, kdežto u kvadrantových stromů roste velmi strmě z důvodů uvedených v předchozích odstavcích.



Závislost velikosti uzlu na dimenzi prostoru 5.1



Poměr velikosti uzlu k velikosti dat v závislosti na dimenzi 5.2

Při rozsahových dotazech se situace z pohledu výkonu obrací a R-strom má zhruba 1,3 krát vyšší výkon, přičemž výkonový rozdíl s rostoucím počtem uzlů nadále vzrůstá.

Nižší výkon R-stromu v případě vkládání a bodových dotazů je dán tím, že minimální ohraničující obdélníky se ve vnitřních uzlech protínají a v případě těchto operací se tedy prochází více cest [5]. Naproti tomu se v kvadrantovém stromu prochází vždy pouze jedna cesta od kořene k listovému uzlu. Naopak je díky této vlastnosti R-strom rychlejší v rozsahových dotazech, protože nalezení relevantních bodů proběhne dříve díky procházení více cest zároveň. Taktéž každý bod kvadrantového stromu je ekvivalentní jednomu uzlu, zatímco u R-stromu se stránky, kde jsou uloženy uzly, procházejí sekvenčně. Tudíž se při práci s uzly kvadrantového stromu nevyužívá L2 cache paměti procesoru a provádí se mnoho náhodných přístupů do paměti.

U kolekcí s reálnými daty je situace odlišná, u kvadrantového stromu se zde prohlubuje problém s výškou stromu. Co se týče vytváření indexu, je R-strom zhruba 3 krát výkonnější. U bodových dotazů je 60 krát výkonnější a u rozsahových dotazů je rychlejší dokonce 144 krát.

Důvodem je, že u této kolekce dat rapidně narůstá výška kvadrantového stromu, body jsou strukturovány poměrně blízko u sebe, třetí souřadnice se mění pouze v rozmezí hodnot 1 až 15. Výška kvadrantového stromu dosahuje hodnoty 2594, zatímco výška R-stromu je pouze 3. Výkon kvadrantového stromu výrazně klesá, protože se musí při každé operaci procházet mnohem více bodů. Dalším důvodem větší rychlosti R-stromů, je nevyváženost kvadrantového stromu, naproti tomu je R-strom vyvážený.

Je vidět, že efektivita použití kvadrantových stromů je dána velikostí dimenze a také povahou indexovaných dat. U středně velké dimenze (5), kde nedochází k velkému nárůstu výšky stromu, je výkon kvadrantových stromů velmi dobrý, co se týče vytváření indexu a bodových dotazů. Taktéž velikost vytvořeného indexu je u menších dimenzí poměrně nízká.

Pokud však potřebujeme indexovat data o větší dimenzi, nebo naopak o velmi nízké dimenzi (2), je lepší použít R-strom. Když R-strom aplikujeme na relativně blízké body, efektivita struktury narůstá. Taktéž velikost indexu je menší. Pokud máme například data o dimenzi 10 s počtem uzlů 1 000 000, potřebujeme u kvadrantových stromů 4 136 MB místa v paměti. U R-stromu se velikost indexu bude pohybovat kolem hodnoty 62,8 MB, což je zásadní rozdíl.

7 Závěr

Cílem diplomové práce byl výzkum a implementace datové struktury kvadrantový strom. Dalším účelem práce byly taktéž výkonové testy a porovnání s jinou vícedimenzionální datovou strukturou.

Princip kvadrantových stromů byl podrobně rozepsán společně s popisem dalších vícedimenzionálních datových struktur.

Implementace byla provedena pomocí frameworku DBRG, ve kterém již jsou implementovány i další vícedimenzionální datové struktury. V práci je pak popsán podrobný způsob implementace, včetně popisu vytvořených tříd a metod. Pro popis funkčnosti algoritmů je použit pseudokód. Při implementaci nastaly drobné problémy s neefektivitou řešení, které byly vyřešeny.

Implementovaný kvadrantový strom byl podroben výkonovým testům a porovnán s podobnou datovou strukturou – R-stromem. Pro testování byly použity data s různou dimenzí a různým počtem bodů.

Nejdříve proběhlo testování pro kolekci s náhodnými daty. Bylo zjištěno, že při dimenzi 5 je kvadrantový strom velmi efektivní datovou strukturou, jejíž výkon se projevuje hlavně při vytváření indexu a bodových dotazech. U této dimenze byl výkon kvadrantového stromu pětkrát vyšší, než výkon R-stromu. Naopak se při dimenzi 2 u kvadrantového stromu projevil problém s výškou stromu, která dosáhla hodnoty 27 pro 1 000 000 uzlů, zatímco u dimenze 5 byla výška kvadrantového stromu 11. Proto se snížila jeho efektivita. Stále však byl o několik procent výkonnější. Naopak R-strom byl výkonnější při rozsahových dotazech a nedocházelo u něj k problému s nárůstem výšky stromu, jelikož se tato výška pohybovala mezi hodnotami 3 a 4. Taktéž má R-strom mnohem menší prostorovou složitost, protože kvadrantový strom musí v každém uzlu uchovávat odkazy na všechny své potomky.

U kolekce reálných dat byly R-stromy výkonnější ve všech případech, protože u kvadrantového stromu narostla výška na hodnotu 2594. Data v této kolekci byly dimenze 3 a třetí atribut měl velmi malou doménu, díky čemuž byly body velmi blízko u sebe. Proto rostla výška takto strmě.

Struktura kvadrantový strom, implementována v rámci této diplomové práce, se bude nadále využívat v rámci frameworku DBRG, který je využíván pro výzkum a testování vícedimenzionálních datových struktur.

Při vypracování a implementaci jsem využil své znalosti programování v jazyce C++ a taktéž všeobecné znalosti vývoje algoritmů získané v předmětu Základy algoritmizace. Novou a cennou zkušeností byla pro mě práce s frameworkem DBRG a jelikož se jedná o obsáhlý aplikační rámec, získal jsem dovednost orientovat se a pracovat s takto rozsáhlou vývojovou platformou. Diplomová práce byla taktéž přínosem pro výzkumnou skupinu DBRG, jelikož výsledek této práce se bude nadále využívat.

Použitá literatura

[1] Samet, Hanan. *Foundations of multidimensional and metric data structures.* Boston: Elsevier/Morgan Kaufmann, 2006, 993 s. ISBN 978-012-3694-461.

[2] LIU, Ling and M. ÖZSU. *Encyclopedia of database systems.* New York: Springer, 2009, ISBN 978-038-7496-160.

[3] Finkel, R. A. a J. L. Bentley. *Quad trees a data structure for retrieval on composite keys.* Acta Informatica, 1974, ISSN 10.1007/BF00288933.

[4] DBRG Framework API. *Database Research Group* [online]. (c) 2001-2012 [cit. 2012-04-18]. Dostupné z: http://db.cs.vsb.cz/api/

[5] Guttman, Antonin. *R-trees: a dynamic index structure for spatial searching*, Proceedings of the SIGMOD Conference, Boston, MA, 1984

[6] Manolopoulos, Yannis. *R-trees: theory and applications*. London: Springer, 2006. Advanced information and knowledge processing. ISBN 978-185-2339-777.

[7] Beckmann N. and Kriegel, H. P. and Schneider, R. and B. Seeger. *The R*-tree: an efficient and robust access method for points and rectangles*, Proceedings of the SIGMOD Conference, Atlantic City, NJ, 1990

[8] Kouba, Zdeněk. *On-line analýza geografické informace* [online]. (c) 2005 [cit. 2012-04-18]. Dostupné z: http://www.cvut.cz/informace-pro-zamestnance/habilitace/hp/hp2003.

[9] Knuth, Donald Ervin. *The art of computer programming*. 3rd ed. Upper Saddle River: Addison-Wesley, 1998. ISBN 02-018-9685-0.

[10] Devroye, Luc. *A note on the height of binary search trees*. Journal of the ACM, 1986

[11] Devroye, Luc. *Branching processes in the analysis of the heights of trees*. Acta Informatica, 1987

[12] Samet, Hanan. *Deletion in two-dimensional quad trees*. Communications of the ACM. 1980

Přílohy

A. Zdrojový kód metody Insert

```
template<class TKey>
uint cQTree<TKey>::Insert(TKey &item, char* data)
{
	uint insertFlag = cQTree::NODE_NOT_INSERTED;
	if (mReadOnly)
	{
		printf("Critical Error: cQTree::Insert(), The tree is read
only!\n");
		exit(1);
	}
	uint nextQuadrant = 0;
	tNodeIndex nodeIndex = mHeader->GetRootIndex();	//first node in
the tree
	TNode *currentNode = NULL;	//pointer, which represents current node
of the tree
	if (!IsInSpace(item))
	{
		return insertFlag;
	}
	while(insertFlag == cQTree::NODE_NOT_INSERTED)
	{
		currentNode = ReadLeafNodeW(nodeIndex);
		if(cQTree::DEBUG == cQTree::DEBUG_INDIVIDUALLY)
		{
			cout << "INSERT>Current node: ";
			currentNode->Print();
		}
		if (currentNode->GetItemCount() == 0)	//if the node is empty
(there is no item), then create new node with inserted item
		{
			currentNode->Insert(item, data, false);
			insertFlag = cQTree::NODE_INSERTED;
		}
		else if (IsInNode(currentNode, item))	//inserted item is the
same like item in currently processed node = data are already located in
the tree
		{
			insertFlag = cQTree::NODE_ALREADY_INSERTED;
		}
		else //inserted item is different than current item => compute
next quadrant
		{
			nextQuadrant = ComputeQuadrant(currentNode,item);
			nodeIndex = currentNode->GetLink(nextQuadrant);
		}
```

```
            // Is the link empty? Create new node and insert the item.
            if (insertFlag == cQTree::NODE_NOT_INSERTED && nodeIndex ==
cNode::EMPTY_INDEX)
            {
                  TNode *newNode = ReadNewLeafNode();
                  newNode->Insert(item, data, false);
                  currentNode->SetLink(nextQuadrant, newNode->GetIndex());
                  mSharedCache->UnlockW(newNode);
                  insertFlag = cQTree::NODE_INSERTED;
            }
            mSharedCache->UnlockW(currentNode);
      }
      return insertFlag;
}
```

B. Adresářová struktura přiloženého DVD

Přiložené médium DVD obsahuje elektronickou verzi diplomové práce ve formátu PDF/A a taktéž zdrojové kódy implementované struktury kvadrantový strom.

Základní struktura DVD:

diplomova_prace – text diplomové práce ve formátu PDF/A

source_code – zdrojové kódy kvadrantového stromu

Seznam příloh